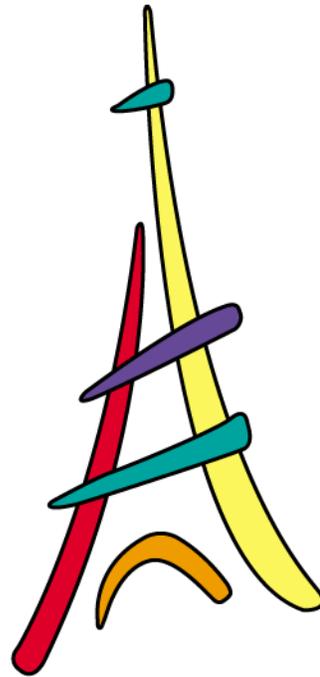


EiffelBench Guided Tour

version 4.3



Eiffel Power™

from ISE

Interactive Software Engineering
eiffel.com

Copyright notice and proprietary information

Copyright ©1999 Interactive Software Engineering Inc. (ISE). May not be reproduced in any form (including electronic storage) without the written permission of ISE. “Eiffel Power” and the Eiffel Power logo are trademarks of ISE.

All uses of the product documented here are subject to the terms and conditions of the ISE Eiffel user license. Any other use or duplication is a violation of the applicable laws on copyright, trade secrets and intellectual property.

Special duplication permission for educational institutions

Degree-granting educational institutions using ISE Eiffel for teaching purposes as part of the Eiffel University Partnership Program may be permitted under certain conditions to copy specific parts of this book. Contact ISE for details.

About ISE

ISE (Interactive Software Engineering) is dedicated to improving software quality and productivity through advanced methods, tools and languages, based on sound scientific principles and on the systematic application of object technology.

The company provides a complete line of development tools as well as on-site consulting, library development services, and a training program on all aspects of O-O technology: analysis, design, implementation techniques, graphics, library building, Eiffel language, project management, large system design etc.

ISE is the original designer of the Eiffel method and language and a member of NICE, the Nonprofit International Consortium for Eiffel.

For more information

Interactive Software Engineering Inc.
ISE Building, 2nd floor
270 Storke Road
Goleta, CA 93117 USA
Telephone 805-685-1006, Fax 805-685-6869

Internet and e-mail

ISE maintains a rich source of information at <http://eiffel.com>, with more than 1200 Web pages including online documentation, downloadable files, product descriptions, links to ISE partners, University Partnership program, mailing list archives, announcements, press coverage, Frequently Asked Questions, Support pages, and much more.

Write to info@eiffel.com for information about products and services. Write to userlist-request@eiffel.com to subscribe to the ISE Eiffel user list.

Support programs

ISE offers a variety of support options tailored to the diverse needs of its customers. Write to info@eiffel.com or check the support pages at <http://eiffel.com> for details.

1

Tutorial: Guided tour

This chapter walks you through the essential properties of EiffelBench. After you read and execute the suggested procedures, you will acquire the basics of working with EiffelBench, including how to:

- Create a new project and retrieve an existing one.
- Add new software elements (classes).
- Compile using **Melting Ice Technology**. For more information, see [“Compiling a system”, page 6](#).
- Execute the result.
- Browse through a software system to view the components and their relationships.

1.1 Before starting

Since people with vastly different backgrounds use and enjoy ISE Eiffel, this chapter only assumes that you can perform basic operations on your platform of choice, such as: use the drag-and-drop operation to move files (Windows) or change directories (Unix, Linux or VMS).

Of course, the more you already know about Eiffel or object technology, the better. However, if you are familiar with other environments, keep an open mind. EiffelBench *is* different and it may take awhile to completely understand why some things are done a certain way.

For the purposes of this example, you will run the tutorial provided in `$EIFFELA/examples|bench|tour`, where `$EIFFELA` is the path for the Eiffel43 directory and `|` is the path separator: backslash (`\`) on Windows or slash mark (`/`) on Unix.

To run the example, you first need to install ISE Eiffel and configure the environment. The environment variable `EIFFELA` must be set to the installation directory, and the environment variable `PLATFORM` to your platform. On Windows this is done automatically by the installation procedure, but on Unix, Linux or VMS you must update your path and environment manually.

Other considerations include:

- On Unix, Linux or VMS, the path must include the location of the EiffelBench executable files.
- Create a directory named *YOURDIR* and then copy all files of the example to it — *YOURDIR* is the name of the directory that you create.
- This discussion also assumes that as part of the installation you included the EiffelBase library, in precompiled form. EiffelBase is automatically provided if you installed another precompiled library, such as the Windows Eiffel Library (WEL).
- Once you compile the example using EiffelBench and precompiled EiffelBase (default), the contents of *YOURDIR* (non-optimized) require 1.25 MB. Without precompiled EiffelBase, 5.5 MB are necessary. In finalized (optimized) mode, the executable needs 300 KB.

1.2 Starting EiffelBench

ISE Eiffel is one of the most portable environments in the industry and runs almost identically on Windows 95/98/NT, Linux, Unix variants and VMS. Consequently, all procedures in this manual apply to all platforms, unless noted otherwise. Nevertheless, how you start EiffelBench depends on the operating system you are using.

Unix, Linux or VMS version

You can start EiffelBench from any directory. However, to simplify things for this tutorial and allow the use of relative rather than absolute file names, you will start EiffelBench from *YOURDIR*.

To start the Unix, Linux or VMS version of EiffelBench:

- Open a shell tool or command window, switch to *YOURDIR*, and then type **ebench**.

The main EiffelBench window appears. For more information on the EiffelBench window, see [“Exploring the Project Tool”, page 3](#).

Windows version

To start the Windows version of EiffelBench:

- 1 On the taskbar, click **Start**, and then point to **Programs**.
- 2 Point to the folder that contains EiffelBench, and then click **EiffelBench**.

The main EiffelBench window appears. For more information on the EiffelBench window, see [“Exploring the Project Tool”, page 3](#).

1.3 Creating an EiffelBench system

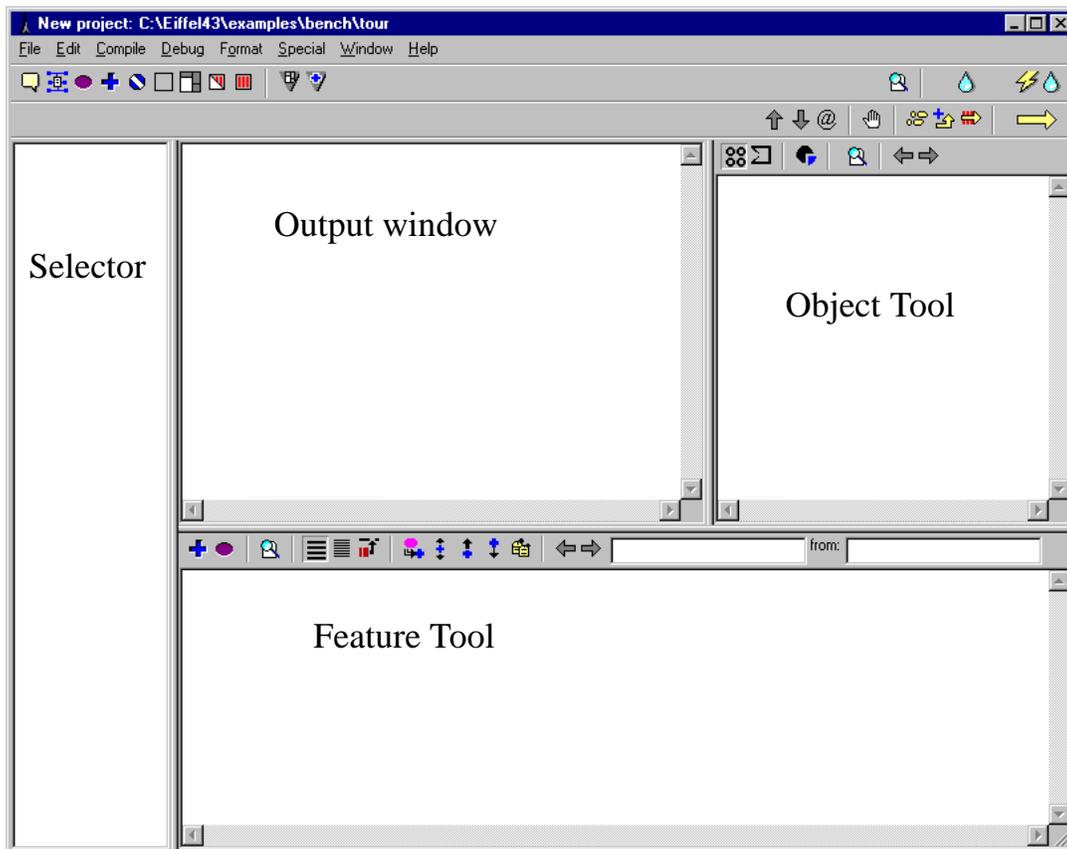
To create the system for the EiffelBench tutorial:

- 1 On the **File** menu, click **New**.
- 2 In the **Directories** list, click the **YOURDIR** folder, and then click **OK**.

When you create an EiffelBench project, an Eiffel generation (**EIFGEN**) subdirectory is added to your EiffelBench system directory. The compiler creates and maintains this subdirectory to store information about your project, and includes the code generated for execution. The path, file name, and view for the active system display in the EiffelBench title bar.

1.4 Exploring the Project Tool

When you start an EiffelBench session, the **Project Tool** displays by default.



The **Project Tool** serves as the control panel for compilation and contains the following:

- **Selector** — lists the targets of all active **Class**, **Feature** and **Object Tools**. One of the following messages display for untargeted tools:
 - Empty class tool
 - Empty feature tool
 - Empty object tool
- **Output window** — displays information about the active project.
- **Object Tool** — displays information about the active object — the object on which execution stopped.

- **Feature Tool** — displays information about the active routine — the one whose execution is in progress.

Ace files

To define a new project, you must provide an Assembly of Classes in Eiffel (Ace) file, which lists the directories for the Eiffel software, external software, and contains the compilation options. An Ace file is written in the Language for the Assembly of Classes in Eiffel (Lace) — a simple, Eiffel-like notation.

An Ace file is required to start the first compilation of a new project. When you open an existing project, there is an associated Ace file. However, for a new project you must either:

- Select an existing Ace file.
- Build an Ace file using EiffelBench.

In this tutorial, you will select an existing Ace file and explore the contents using the **System Tool**.

1.5 Exploring the System Tool

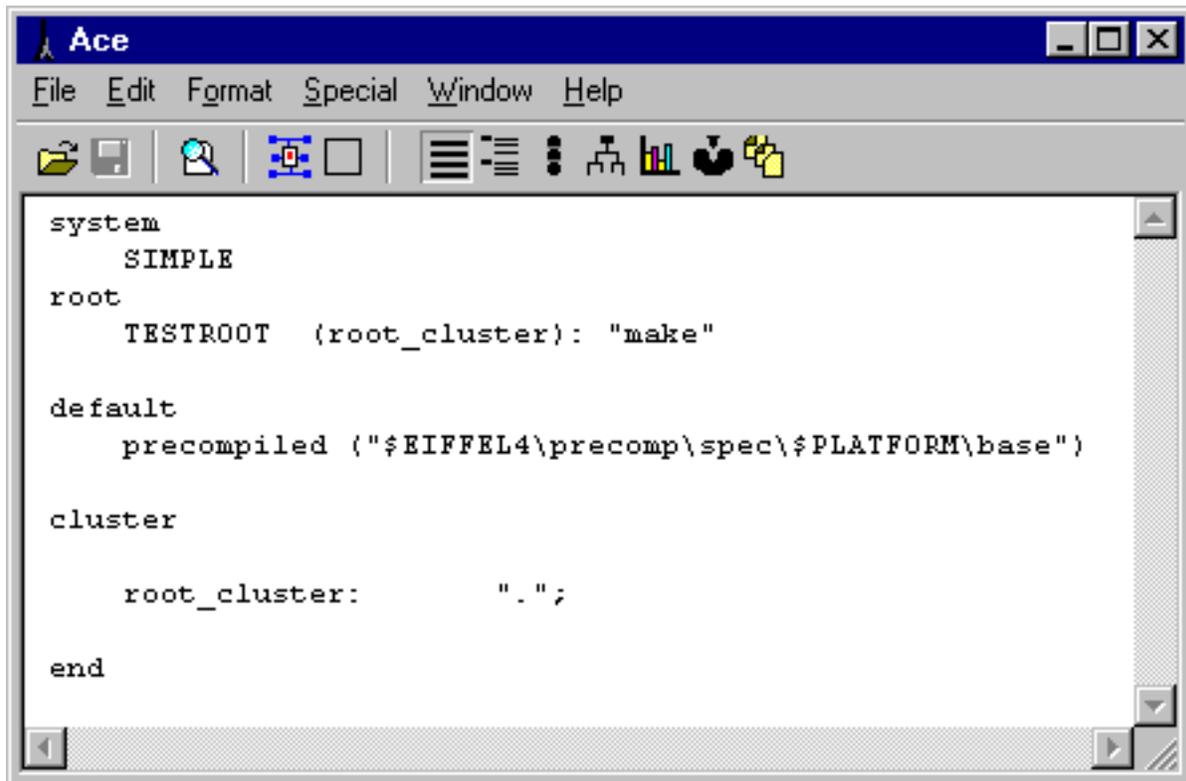
The **System Tool** defines the overall structure a system, and accesses all clusters. It also creates, defines and modifies Ace files.

Selecting an existing Ace file

To select an existing Ace file:

- 1 On the **Project Tool** toolbar, click **System** .
- 2 Click **Browse**, in the **Look in** list, click Ace.ace, and then click **Open**.

The **System** Tool appears and displays the Ace file, which contains the following clauses in Lace syntax:

A screenshot of the Ace editor window. The title bar reads "Ace". The menu bar includes "File", "Edit", "Format", "Special", "Window", and "Help". The toolbar contains icons for file operations, search, and editing. The main text area contains the following Lace syntax:

```
system
  SIMPLE
root
  TESTROOT (root_cluster): "make"

default
  precompiled ("EIFFEL4\precomp\spec\PLATFORM\base")

cluster

  root_cluster:      ".";

end
```

- **system** — the name of the active system (**SIMPLE**).
- **root** — the name of the root class (**TESTROOT**), the cluster to which **TESTROOT** belongs (**root_cluster**), and the name of the creation procedure for **TESTROOT** (**make**).
- **default** — contains a **precompiled** entry that indicates that the active project uses a precompiled version of the EiffelBase library, stored in the specified path.
- **cluster** — lists all clusters in the active system by name, followed by a colon and then the name of the associated directory, surrounded by quotation marks. In this example, “.” means that **root_cluster** is in the active directory.

You can now compile the active system.

1.6 Compiling a system

The EiffelBench relies on **Melting Ice Technology**, the proprietary compilation mechanism of ISE Eiffel, which offers three forms of compilation:

- **melting** — for making a few changes. The fastest of the mechanisms, typically taking a few seconds after small changes. Melting time is proportional to the size of the changed parts and affected classes, while the time needed to freeze or finalize is partly proportional to the size of the whole system. As long as you do not include new external C/C++ code, a C/C++ compiler is not required. However, execution speed is not optimal.
- **freezing** — generates C code from the active system, and then compiles it into machine code; you must have a C/C++ compiler installed. You need to use this option if you add external C/C++ software. Unless you add external code, you can re-freeze every couple of days. The rest of the time, you can melt your software to receive immediate feedback.
- **finalizing** — delivers a production version (intermediate or final) of your software or to measure its performance in operational conditions. Finalizing performs extensive time and space optimizations that enable Eiffel to match the efficiency of C/C++; it also creates a stand-alone C package that you can use for cross development. Because of all the optimizations involved, finalizing takes the most time.

Since this example relies on precompiled libraries that contain frozen code, you will use melting throughout the tutorial.

Melting the project

To melt the project:

- On the Toolbar, click **Melt** .

The **Compilation Progress** dialog box displays the percentage of compilation completed.

During compilation, EiffelBench analyzes your system and determines what needs to be recompiled — in this case, all classes, since your system is a new one. *Degrees* measure progress, decrementing from 6 to 1. At degree zero, the changes melt or are effected.

Because many of the classes included in this example are part of EiffelBase and have been precompiled, EiffelBench only compiles the classes in the **root_cluster** directory: **TESTROOT**, **HEIR** and **PARENT**.

The precompilation mechanism plays an important role in the speed of compilation. Although only three classes are compiled, the actual size of the system — the number of classes on which the root class depends directly or indirectly, including **STD_FILES** (used for input and output) and all ancestors and suppliers — is 65 classes.

Apart from the speed, the most important feature of melting is that it is entirely automatic. The only information the compiler retrieves from the Ace file is the name and path for the root class. You do not have to supply “make files” or other manual descriptions of intermodule dependencies.

1.7 User interface conventions

The following section provides a basic overview of the user interface conventions of ISE Eiffel. These conventions apply to all the tools in the environment, including EiffelCase and EiffelBuild.

Development objects

EiffelBench provides a series of tools that work directly with the conceptual units of your project — **development objects**.

Using the ISE Eiffel environment, you work directly with the development objects, while the tools address the underlying source text files. This allows you to concentrate on building the proper logical structure.

Development objects that EiffelBench supports include:

- **Project** — defined by a directory that stores project-related files, and where the compiler stores its internal files in an **EIFGEN** subdirectory.
- **System** — group of classes, spread over a number of clusters; known as a program in traditional approaches.
- **Cluster** — group of classes in the same directory.
- **Class** — set of data abstractions.
- **Features** — set of properties attached to a class.
- **Execution objects** — run-time objects created during execution as instances of classes in the system.

A development object is a concept of ISE Eiffel and is an instance of any of the abstractions. For the purposes of this user guide, the term development object represents either a project, system, class, feature or execution object.

Other ISE Eiffel tools, such as EiffelCase and EiffelBuild, use these types and add several others.

Pick-and-drop operation

The pick-and-drop operation is instrumental in the ISE Eiffel environment. You will use this mechanism throughout this tutorial and the ISE Eiffel environment.

To use the Eiffel pick-and-drop operation:

- 1 Right-click a development object name.

The cursor changes to the shape of the selected object — a **pebble** — and a line continuously connects the pebble to the original position of the object.

- 2 Point to the new position for the object, and then right-click.

Canceling the operation

To cancel the operation:

- Click anywhere.

Holes

Holes are icons that you drop pebbles on using the pick-and-drop operation, to create new tools, retarget existing tools, or execute other operations. The pebble and hole in a pick-and-drop operation must be compatible — not necessarily identical.

There are two types of holes in EiffelBench:

- **Tool holes** — a symbol for the corresponding development abstraction; located in the upper left corner of a tool.
- **Operation holes** — performs various operations on the target of a tool, when you drop the object on the hole.

If you drop a pebble on its corresponding hole, a new tool appears and displays information about the selected development object. This action makes the object the *target* of the new tool — the tool is *targeted* to the object. When you target a tool, the corresponding hole displays with a dot in it.

Since the dot represents the development object; you can use the pick-and-drop operation to drop the dotted hole on its corresponding hole to display a new tool targeted to that class.

Clickable elements

A **clickable element** is one on which you use the pick-and-drop operation, or hold down CTRL and either click or right-click. This definition includes class names, routine declarations (features), stop points and execution objects.

Clickable elements display underlined and in blue on Windows. Because of limitations in the current Motif library, these elements are not distinguished graphically in Unix or VMS.

In the next section, you will use the pick-and-drop operation to target a tool (the **Class Tool**), and then view contents of a class (**TESTROOT**).

1.8 Exploring the Class Tool

The **Class Tool** sets properties, features, indexing information and constraints for the active class.

This section explains three ways to create a **Class Tool** targeted to a specific class:

- Use the pick-and-drop operation.
- Hold down CTRL and right-click a class.
- Hold down CTRL and click a class.

Targeting the Class Tool using the pick-and-drop operation

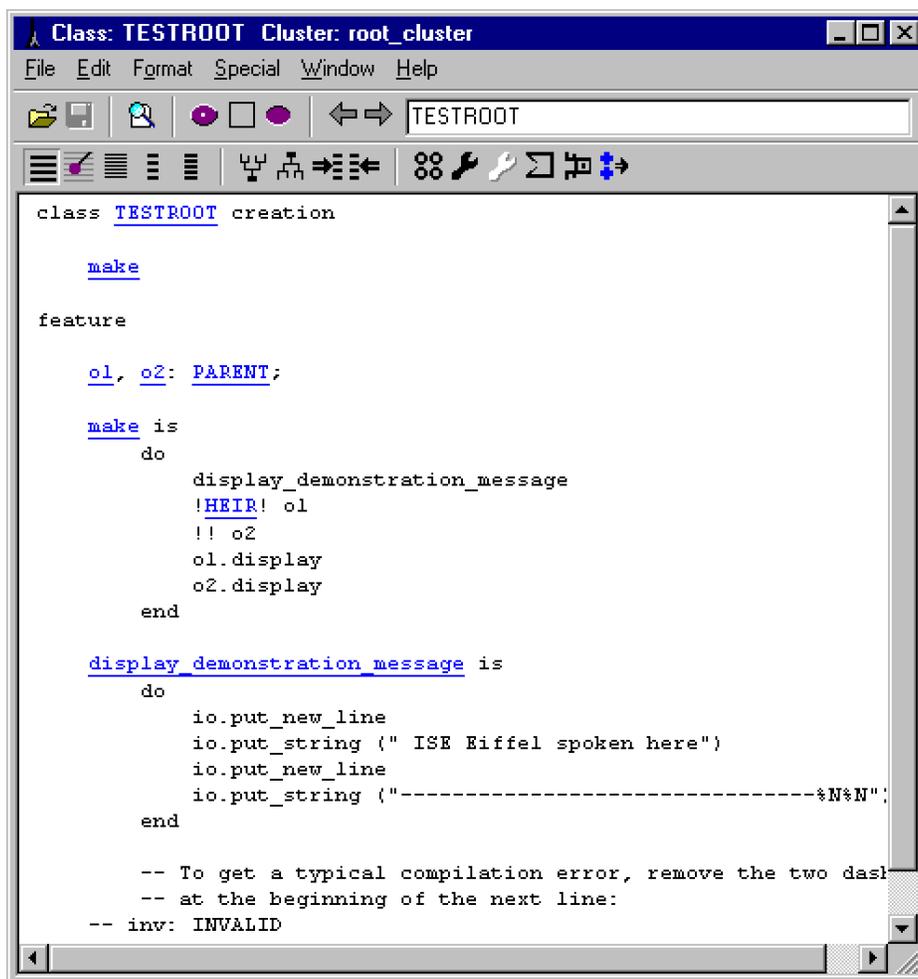
To target the **Class Tool** using the pick-and-operation:

- 1 In the **System Tool** window, right-click **TESTROOT**.

The cluster changes to the shape of the selected object — a class.

- 2 On the **Project Tool** toolbar, point to the **Class** hole , and then right-click.

The **Class Tool** appears and displays the contents of **TESTROOT**.



```

class TESTROOT creation

    make

feature

    o1, o2: PARENT;

    make is
    do
        display_demonstration_message
        !HEIR! o1
        !! o2
        o1.display
        o2.display
    end

    display_demonstration_message is
    do
        io.put_new_line
        io.put_string (" ISE Eiffel spoken here")
        io.put_new_line
        io.put_string ("-----%N%N");
    end

    -- To get a typical compilation error, remove the two dashes
    -- at the beginning of the next line:
    -- inv: INVALID
  
```

Targeting the Class Tool class using CTRL right-click

To target the **Class Tool** using CTRL right-click:

- In the **System Tool** window, hold down CTRL, and then right-click **TESTROOT**.

Targeting the Class Tool using CTRL click

To target the **Class Tool** using CTRL click:

- In the **System Tool** window, hold down CTRL and click **TESTROOT**.

As mentioned earlier, you can also use the pick-and-drop operation to drop a development object on a targeted hole to retarget that hole.

Since there are three open **Class Tools** targeted to **TESTROOT**, you need to close two.

Closing a Class Tool

To close a **Class Tool**:

- On the **File** menu, click **Exit tool**.

In the next section, you will explore the different ways to retarget the **Class Tool**.

1.9 Retargeting the Class Tool

This section introduces the different ways to retarget the **Class Tool**:

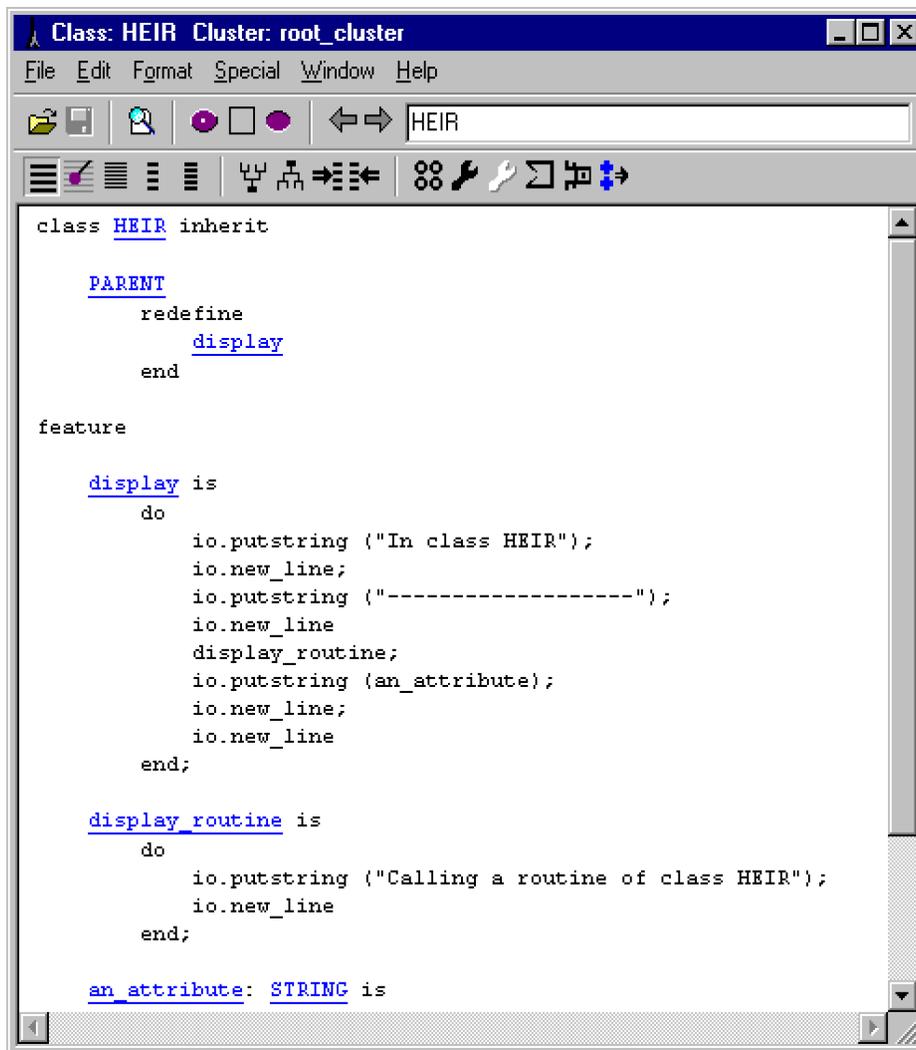
- Use the **Class hole**  on the **Project Tool** toolbar.
- Use the tool window.
- Use browsing accelerators (**Previous**  or **Next**  on the **Class Tool** toolbar).
- Use the **Target Name Tool**.

Using the Class hole

To retarget the **Class Tool** using the **Class** hole:

- 1 In the **Class Tool** window, right-click **HEIR**.
- 2 On the **Class Tool** toolbar, point to the targeted **Class** hole , and then right-click.

The **Class Tool** retargets and displays the contents of **HEIR**.



Before you continue, it is important to note the difference between the preceding pick-and-drop operations:

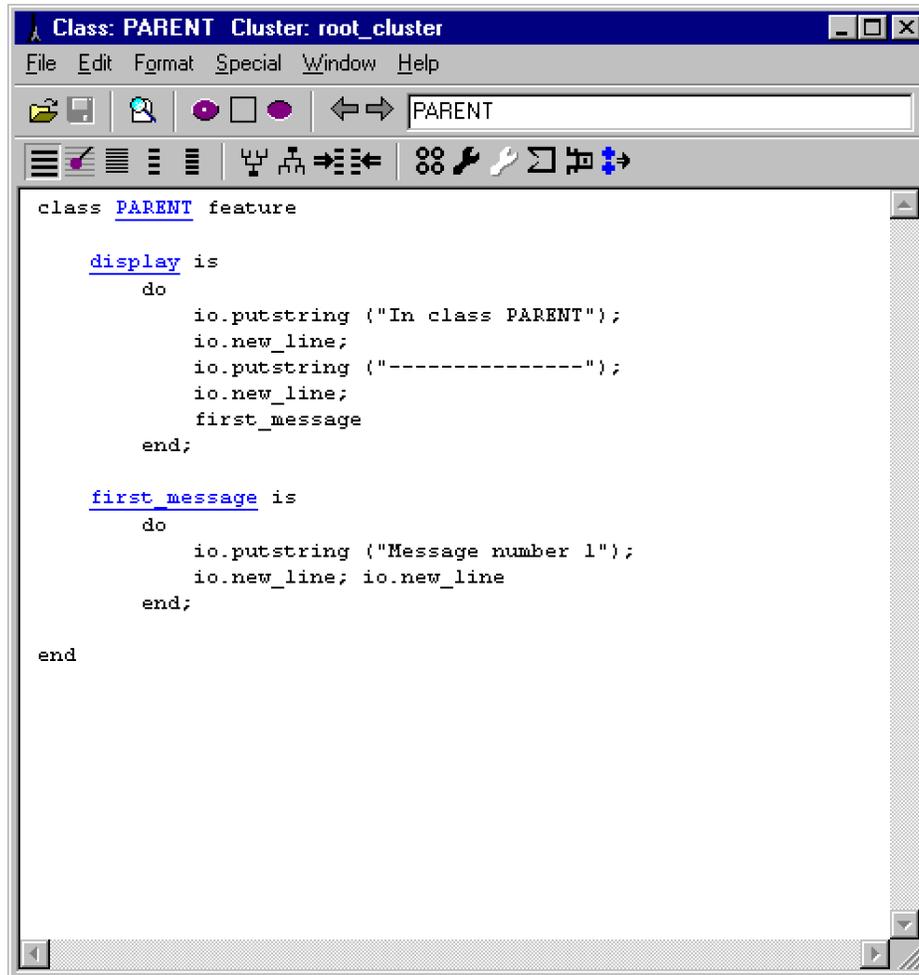
- You create a **Class Tool** targeted to a specific class when you use the pick-and-drop operation to drop a class pebble on the **Class** hole in the **Project Tool** toolbar.
- You retarget an existing **Class Tool** to a specific class when you use the pick-and-drop operation to drop a class pebble on the targeted **Class** hole in the **Class Tool** toolbar.

Using the tool window

To retarget the **Class Tool** using the tool window:

- 1 In the **Class Tool** window, right-click **PARENT**.
- 2 Point anywhere in the **Class Tool** window, and then right-click.

The **Class Tool** retargets and displays the contents of **PARENT**.



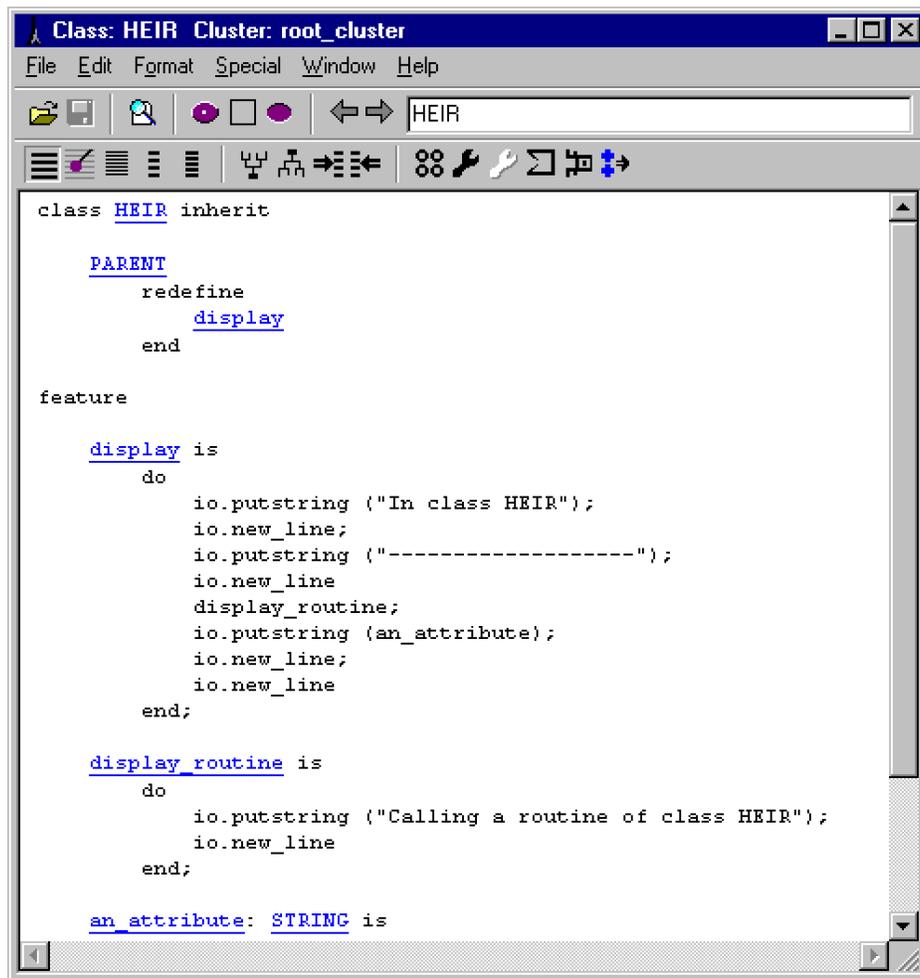
Using browsing accelerators

The browsing accelerators — **Previous**  and **Next**  — are located in the **Project Tool**, **Class Tool**, **Feature Tool** or **Object Tool** toolbars. Since each tool maintains a list of its recent targets, you can also use these commands to go back and display the previous target, display the next target (if applicable).

To retarget the **Class Tool** using the tool window:

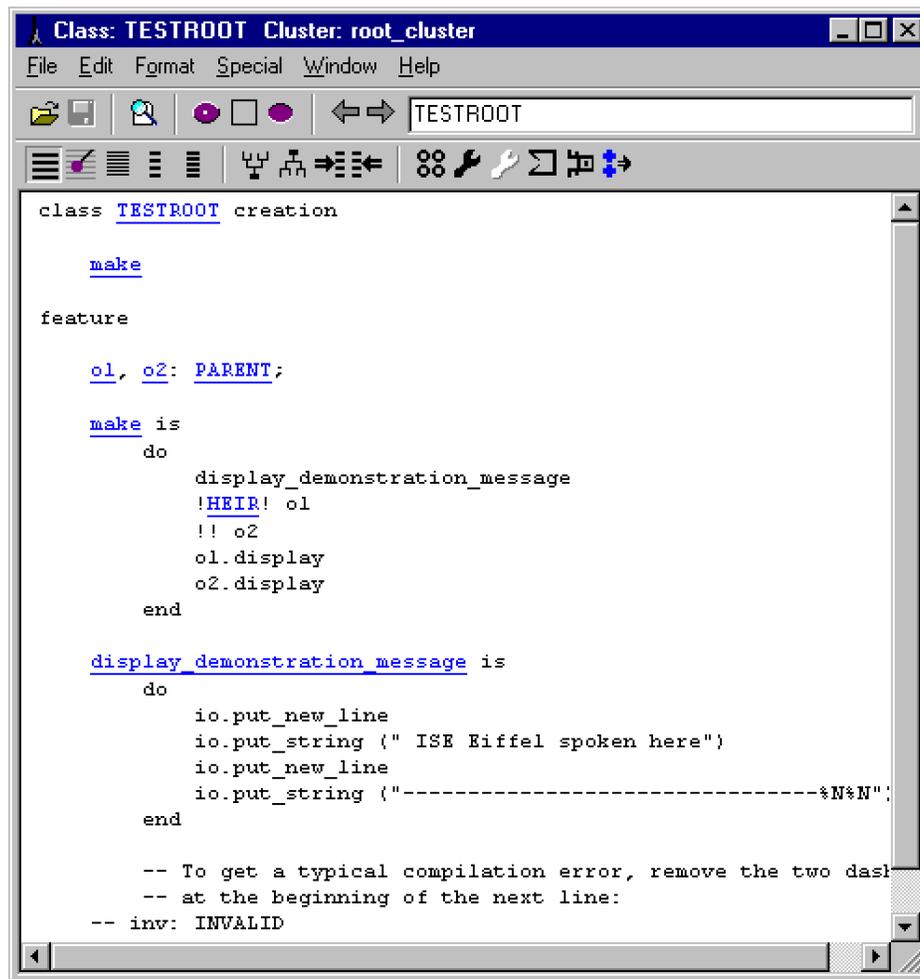
- 1 On the **Class Tool** toolbar, click **Previous**.

The **Class Tool** retargets and displays the contents of **HEIR**.



- 2 Click **Previous** a second time.

The **Class Tool** retargets and displays the contents of **TESTROOT**.



3 Click **Next**.

The **Class Tool** retargets and displays the contents of **HEIR**.

4 Click **Next** a second time.

The **Class Tool** retargets and displays the contents of **PARENT**.

You can also right-click **Previous** or **Next** to display a list of all active targets.

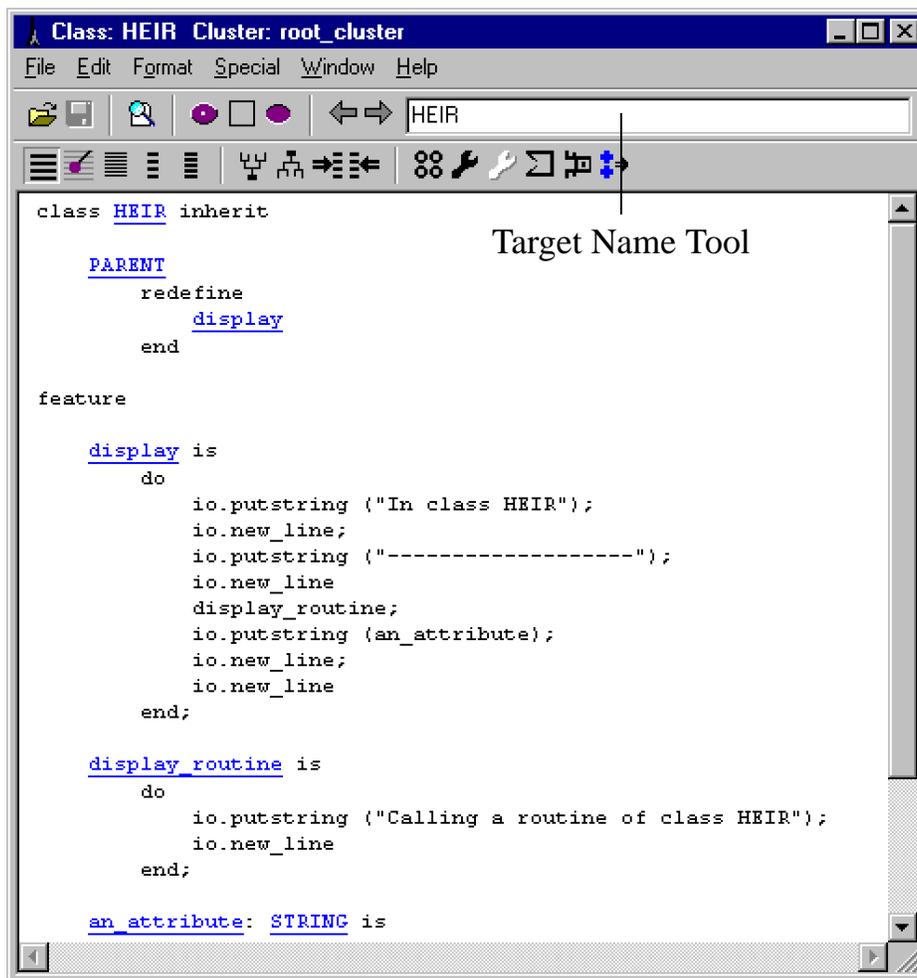
Using the Target Name Tool

The **Target Name Tool** is on the **Class Tool** toolbar and displays the name of the active class, which you can rename. You can also use it to quickly retarget the **Class Tool**.

To retarget the **Class Tool** using the **Target Name Tool**:

- In the **Target Name Tool** box, type **HEIR**, and then press ENTER.

The **Class Tool** retargets and displays the contents of **HEIR**.



Since Eiffel is not case-sensitive, you can use lower case or upper case text when you type. Once the tool retargets, the class name displays by default in upper case in the **Target Name Tool**.

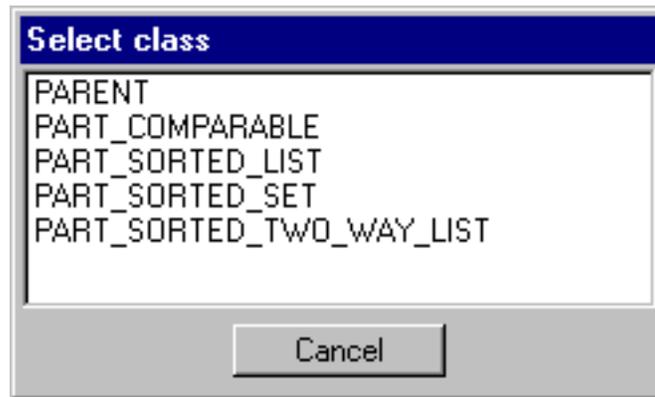
You can also use the wildcard character (*) to represent one or many characters.

Using the wildcard character

To use the wildcard character:

- 1 In the **Target Name Tool** box, type **par***, and then press ENTER.

The **Select class** dialog box appears and lists all classes in the system that begin with par.



- 2 Click **PARENT**.

In the next section, you will display a **Feature Tool** targeted to **display**, and then retarget the tool to explore other features.

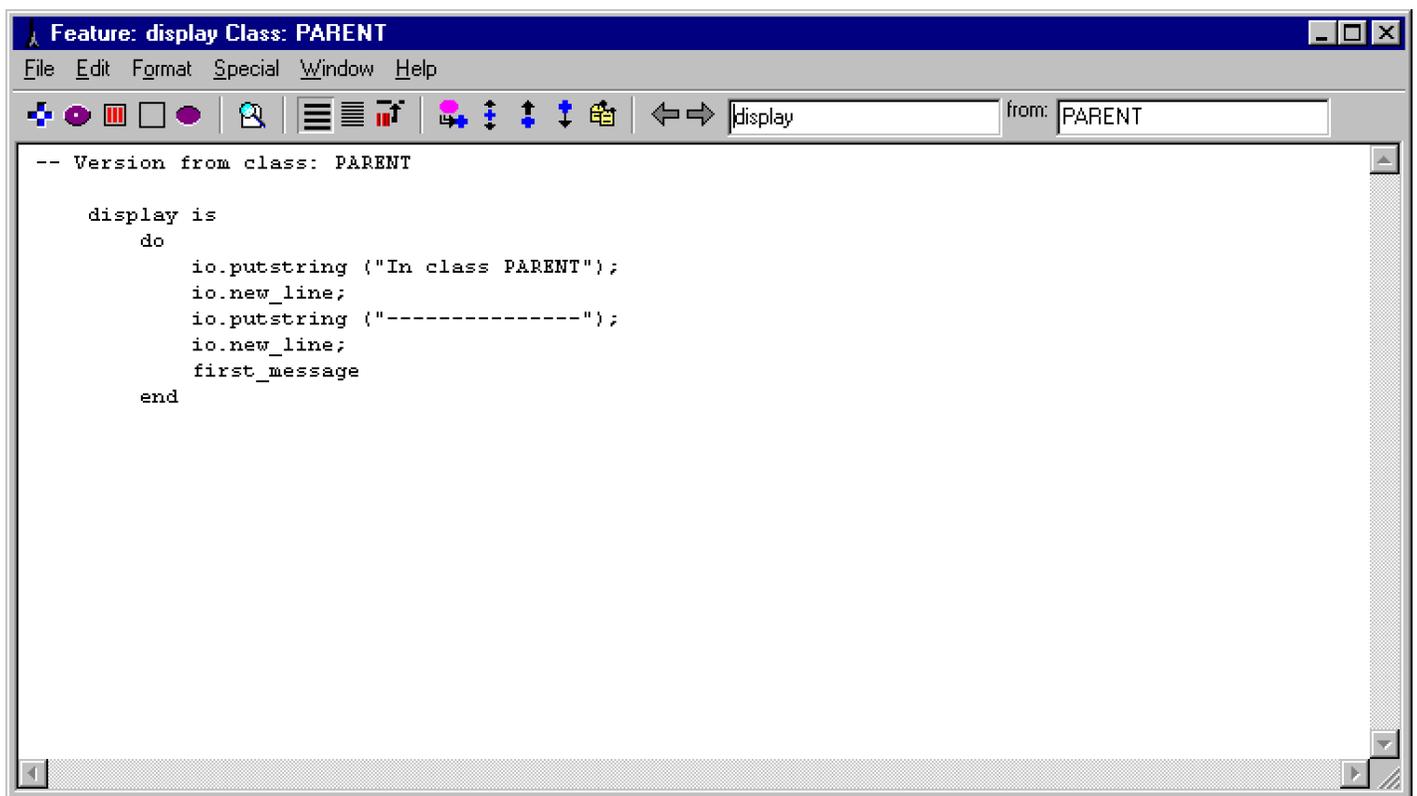
1.10 Exploring the Feature Tool

The **Feature Tool** sets properties for the active feature and its signature — the number and type of arguments in the feature.

Displaying the Feature Tool

To display the **Feature Tool**:

- In the **Class Tool** window, hold down CTRL, and then right-click **display**.



The comment, added by EiffelBench, indicates where the version of **display** is declared, in this case, in class **PARENT**. In general, the applicable version of any feature in any class can come from any ancestor.

The next section examines three ways to retarget the **Feature Tool**:

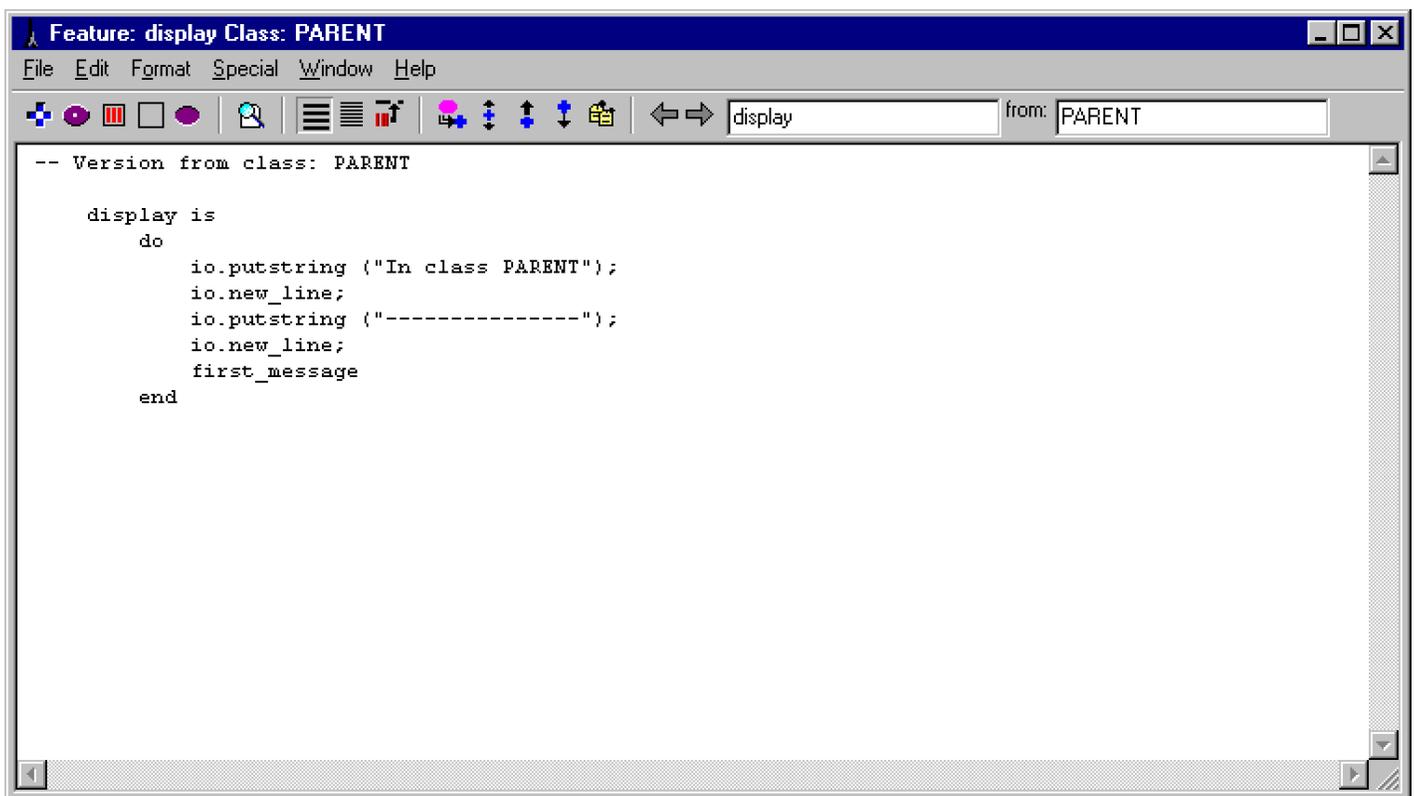
- Use the **Feature hole**  on the **Feature Tool** toolbar.
- Use a class (**HEIR**, one of the descendents of **PARENT**).
- Use the **Target Name Tool**.

Using the Feature hole

To retarget the **Feature Tool** using the **Feature** hole:

- 1 In the **Class Tool** window (targeted to **PARENT**), right-click **display**.
- 2 On the **Feature Tool** toolbar, point to the **Feature** hole , and then right-click.

The **Feature Tool** retargets and displays the version of **display** targeted to **PARENT**.



The screenshot shows a window titled "Feature: display Class: PARENT". The window has a menu bar with "File", "Edit", "Format", "Special", "Window", and "Help". Below the menu bar is a toolbar with various icons. The main area of the window displays the following code:

```
-- Version from class: PARENT

display is
do
  io.putstring ("In class PARENT");
  io.new_line;
  io.putstring ("-----");
  io.new_line;
  first_message
end
```

You can also drop **display** anywhere in the **Feature Tool** window to produce the same results.

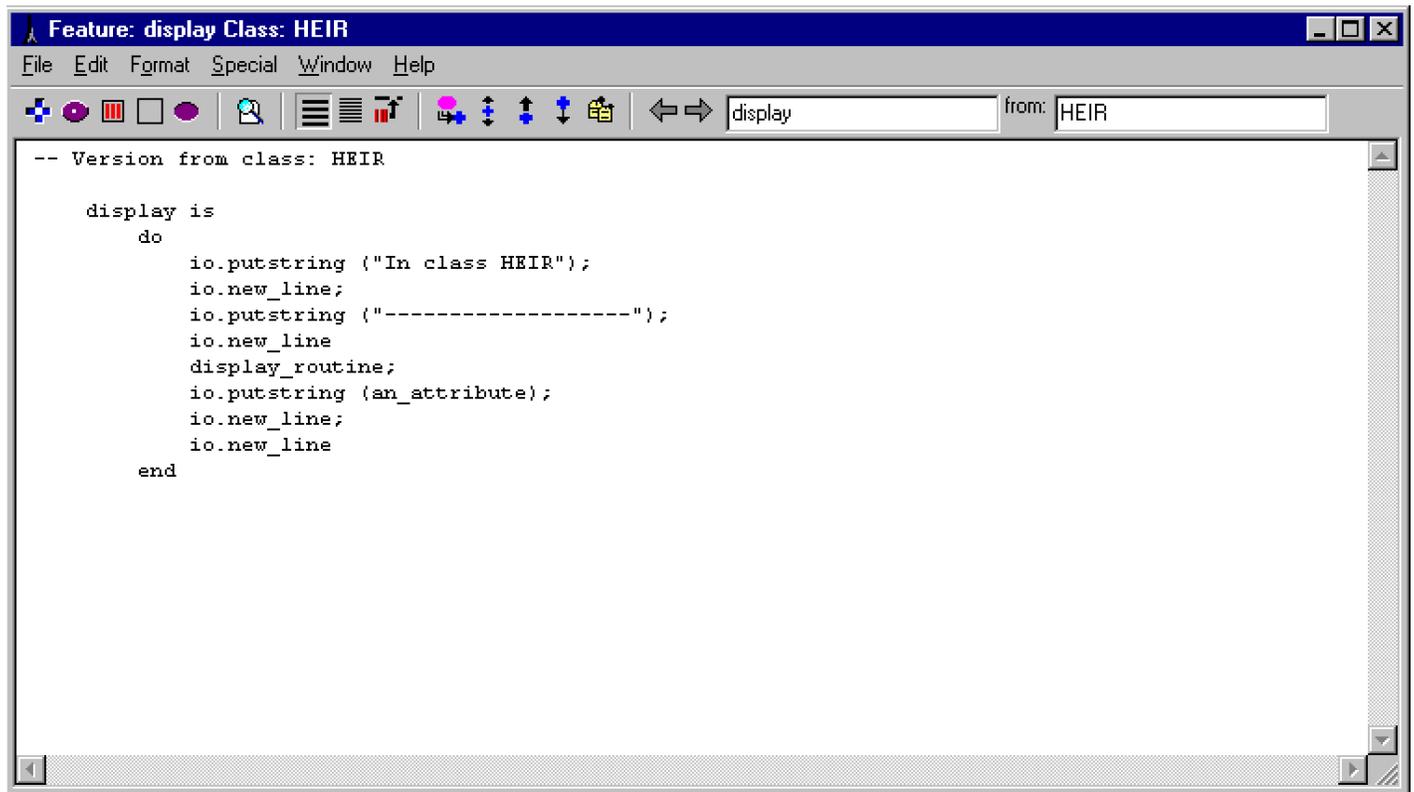
Using a class

To retarget the **Feature tool** using a class:

- 1 On the **Class Tool** toolbar (targeted to **PARENT**), click **Descendents** .
- 2 In the **Class Tool** window, right-click **HEIR**.

3 In the **Feature Tool** window, point anywhere, and then right-click.

The **Feature Tool** retargets and displays the version of **display** defined in **HEIR**.



You will use this mechanism when you know about a feature **f** from class **A**, displaying in a **Feature Tool**, and want to know what the version of **f** is for class **B**. You can use the pick-and-drop operation to drop either a feature or a class on a **Feature Tool**.

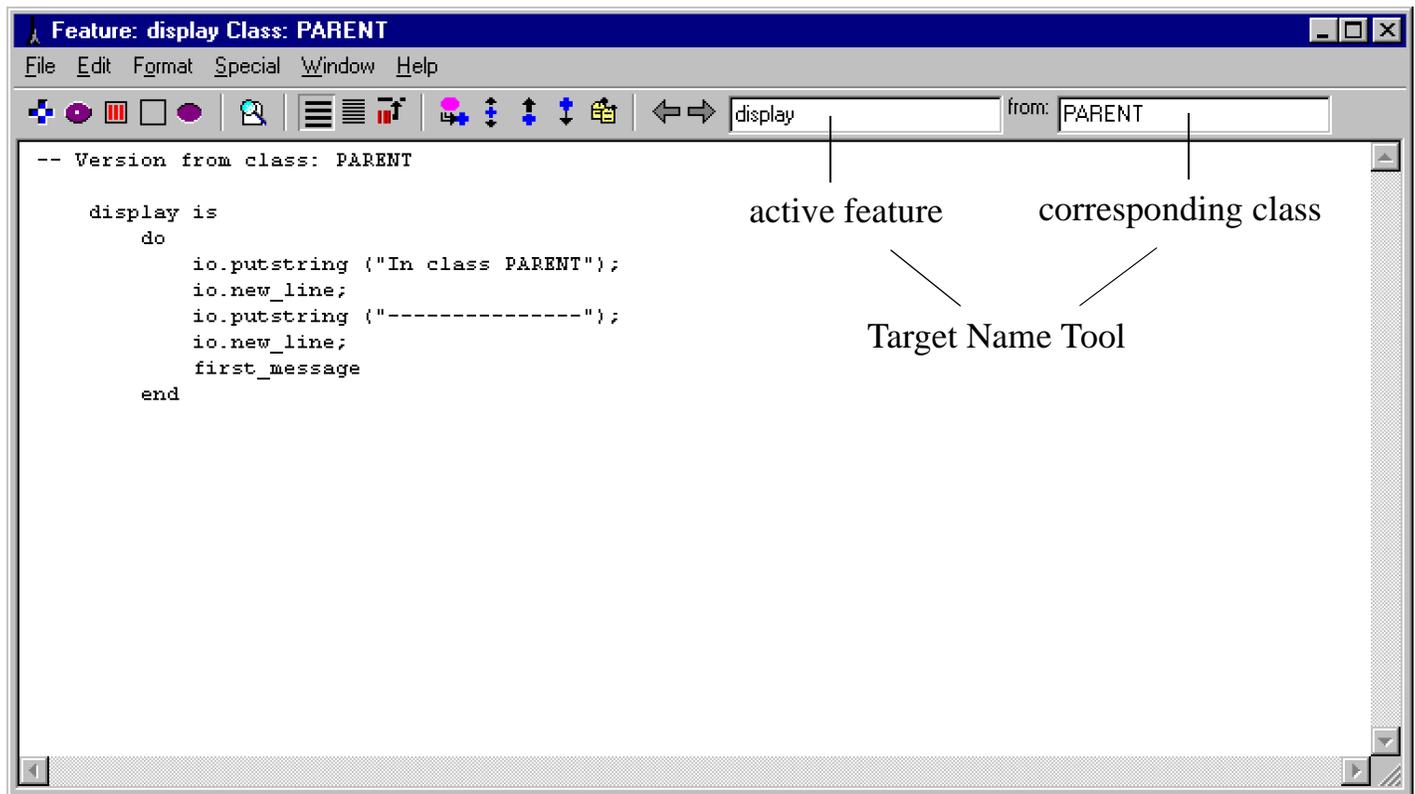
Using the Target Name Tool

Like the **Class Tool**, the **Feature Tool** toolbar also contains a **Target Name Tool**, with a slight modification — two text boxes, instead of one. The left text box displays the name of the active feature, while the right one displays the name of the corresponding class. Functionality remains the same: you can rename the active feature or class, and quickly retarget the tool.

To retarget the **Feature Tool** using the **Target Name Tool**:

- In the **Feature Tool**, in the right **Target Name Tool** box, type **PARENT**, and then press ENTER.

The **Class Tool** retargets and displays the contents of **display** targeted to **PARENT**.



1.11 Formats

In the preceding sections, you have seen the **System Tool**, the **Class Tool** and the **Feature Tool**, with the corresponding Ace file, class and feature text. You can display other information or **formats** about the corresponding development object, such as the classes of the system or the ancestors of a class. Once you retarget a tool, it retains the format displaying prior to retargeting.

This section details the formats available in the **System Tool**, **Class Tool** and **Feature Tool**. In each of the following sections, you will explore the active system, class **TABLE** and several features of the class, respectively.

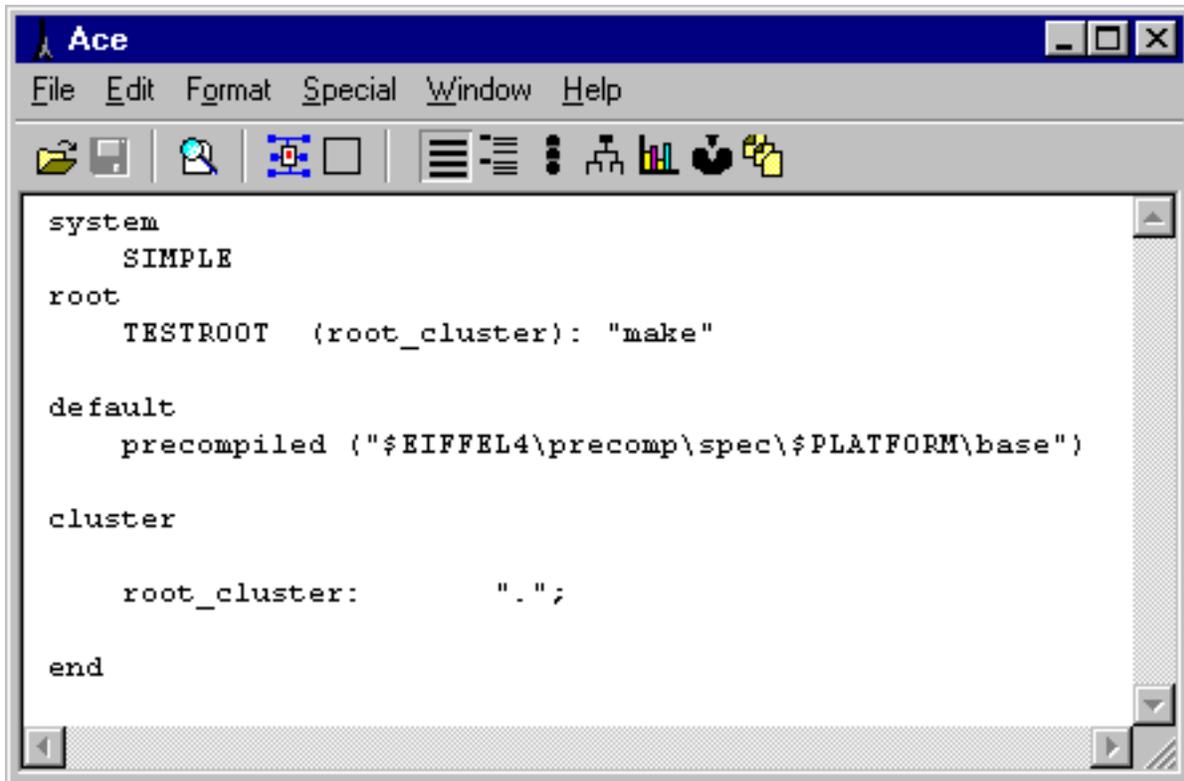
The first series of formats that you will explore are in the **System Tool**. You may need to redisplay this tool.

To redisplay the **System Tool**:

- On the **Project Tool** toolbar, click **System** .

System Tool formats

By default, the **System Tool** appears displaying the Ace file text in the tool window. You can use the commands on the **System** toolbar to view the contents of the active system in different formats.

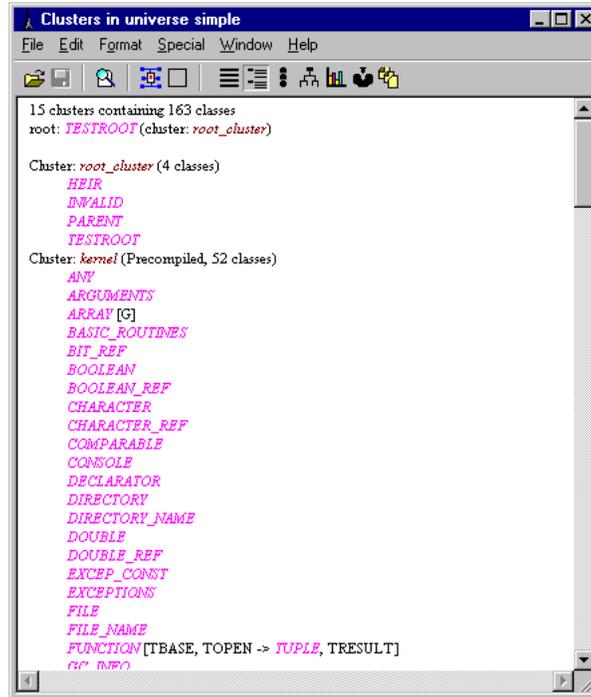


Since all class names in the following formats are clickable, you can hold down CTRL, and then right-click a class name to display its properties in a **Class Tool**.

To view a format below, click the corresponding button on the **System** toolbar.

- **Text** : displays a textual representation of the Ace file for the active system (default — see preceding illustration). In this format, only class **TESTROOT** is clickable.

- **Clusters** : lists all clusters in the active system; all corresponding classes display alphabetically below each cluster; the root cluster (**root_cluster**) always displays first, the precompiled clusters last.



```

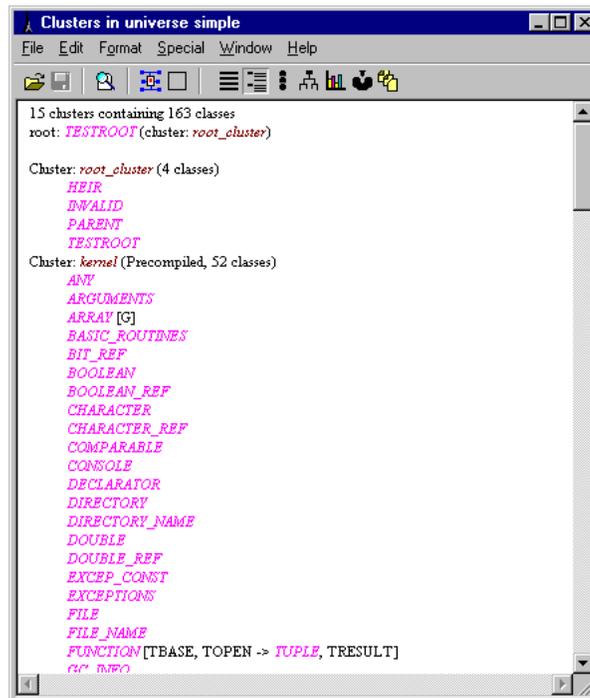
Clusters in universe simple
File Edit Format Special Window Help
15 clusters containing 163 classes
root: TESTROOT (cluster: root_cluster)

Cluster: root_cluster (4 classes)
HEIR
INVALID
PARENT
TESTROOT

Cluster: kernel (Precompiled, 52 classes)
ANY
ARGUMENTS
ARRAY [G]
BASIC_ROUTINES
BIT_REF
BOOLEAN
BOOLEAN_REF
CHARACTER
CHARACTER_REF
COMPARABLE
CONSOLE
DECLARATOR
DIRECTORY
DIRECTORY_NAME
DOUBLE
DOUBLE_REF
EXCEP_CONST
EXCEPTIONS
FILE
FILE_NAME
FUNCTION [TBASE, TOPEN -> TUPLE, TRESULT]
GC_INFO

```

- **Classes** : alphabetically lists all classes in the active system with its corresponding cluster.



```

Clusters in universe simple
File Edit Format Special Window Help
15 clusters containing 163 classes
root: TESTROOT (cluster: root_cluster)

Cluster: root_cluster (4 classes)
HEIR
INVALID
PARENT
TESTROOT

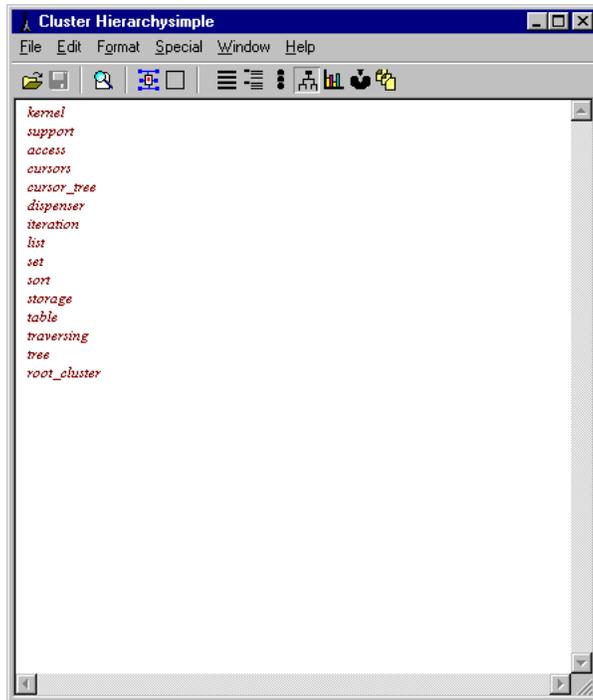
Cluster: kernel (Precompiled, 52 classes)
ANY
ARGUMENTS
ARRAY [G]
BASIC_ROUTINES
BIT_REF
BOOLEAN
BOOLEAN_REF
CHARACTER
CHARACTER_REF
COMPARABLE
CONSOLE
DECLARATOR
DIRECTORY
DIRECTORY_NAME
DOUBLE
DOUBLE_REF
EXCEP_CONST
EXCEPTIONS
FILE
FILE_NAME
FUNCTION [TBASE, TOPEN -> TUPLE, TRESULT]
GC_INFO

```

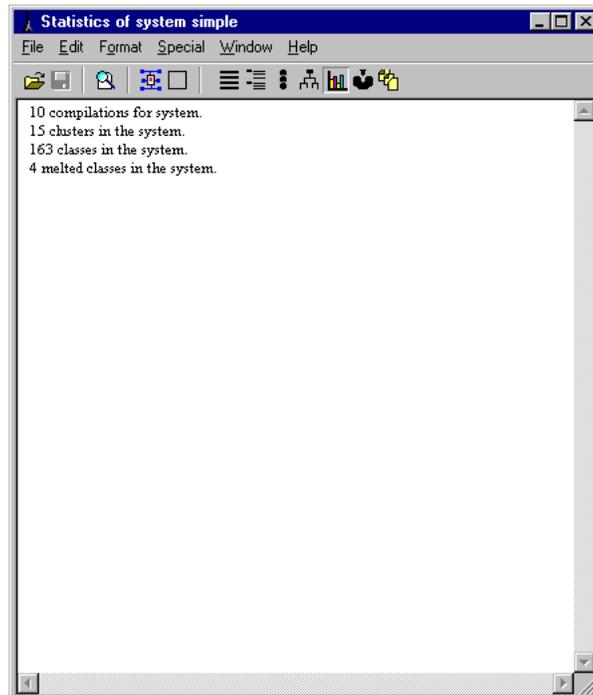
- **Show cluster hierarchy** : displays the cluster hierarchy for the active system. You use the following Lace syntax to specify that one cluster is a subcluster of another:

my_subcluster (some_parent_cluster): "\$/my_subcluster_directory"

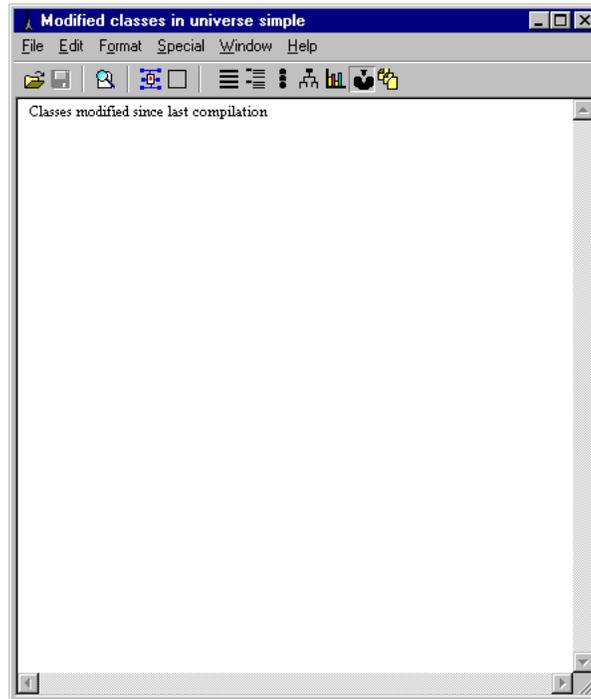
\$/my_subcluster_directory is the directory for the parent cluster *some_parent_cluster*.



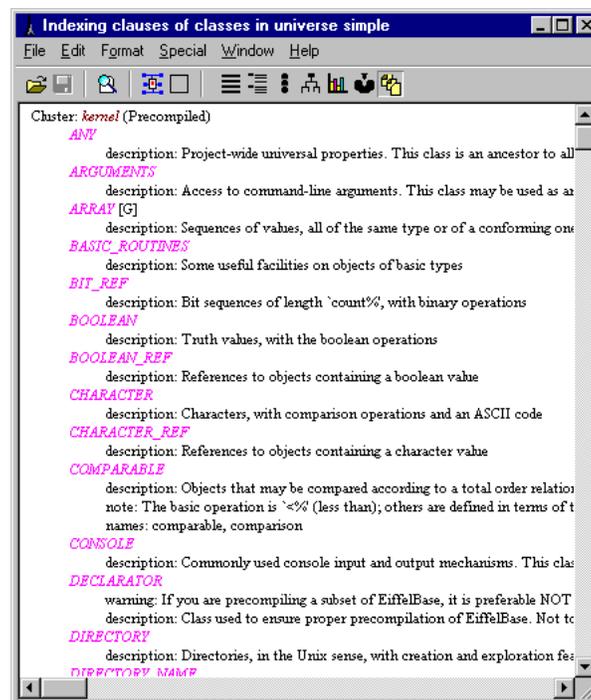
- **Statistics** : lists statistical information for the active system, such as the number of classes and clusters.



- **Modified classes** : lists all classes in the active system that were modified since the last compilation.



- **Indexing clauses** : alphabetically lists all clusters in the active system with key information from the indexing clauses of all classes — notably the **description** entry. The indexing clause displays in the beginning of an Eiffel class and associates indexing information with the class.



The set of formats that you will explore in the next section are in the **Class Tool**, retargeted to class **TABLE** (an EiffelBase class). You may need to redisplay this tool.

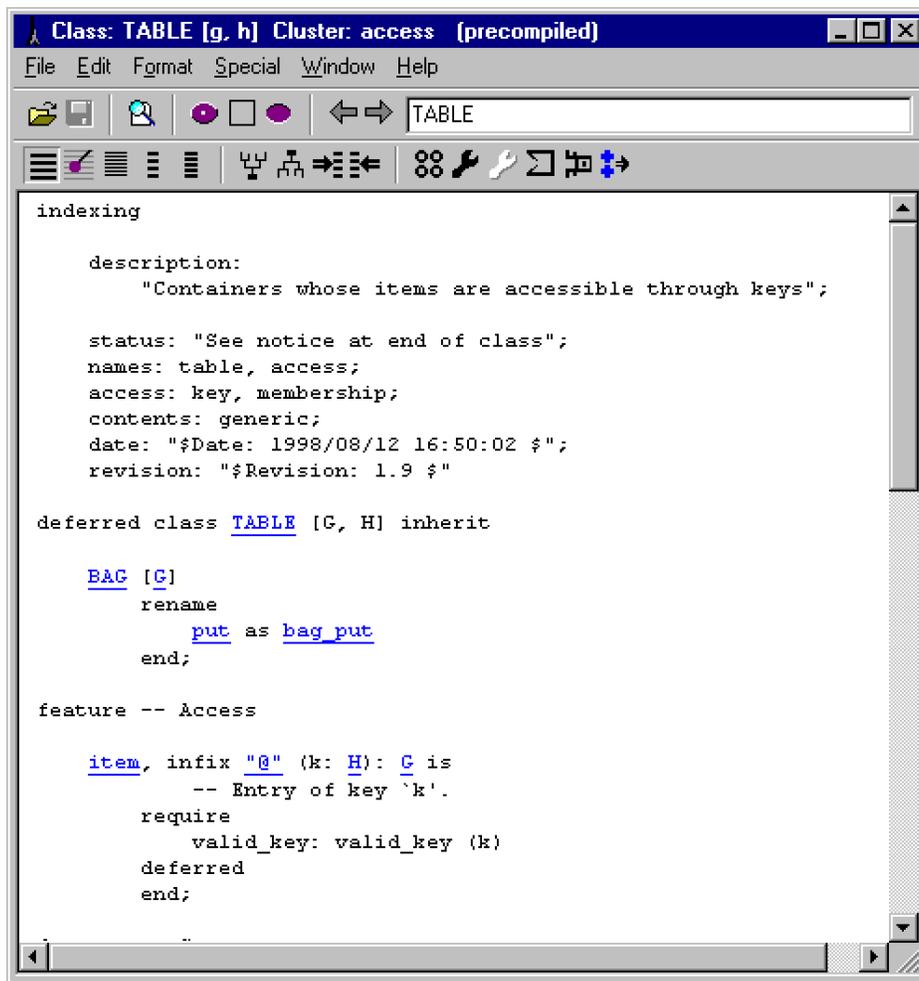
To redisplay the **Class Tool**:

- On the **Project Tool** toolbar, click **Class** .

Before continuing, you must retarget the **Class Tool** to **TABLE**. You can use any of the methods described earlier in this chapter to retarget the **Class Tool**.

Class Tool formats

Like the **System Tool**, the **Class Tool** appears displaying the text for the active class in the tool window. You can use the commands on the **Class** toolbar to view the contents of the active class (**TABLE**) in different formats.



```

indexing

description:
    "Containers whose items are accessible through keys";

status: "See notice at end of class";
names: table, access;
access: key, membership;
contents: generic;
date: "$Date: 1998/08/12 16:50:02 $";
revision: "$Revision: 1.9 $"

deferred class TABLE [G, H] inherit

    BAG [C]
        rename
            put as bag_put
        end;

feature -- Access

    item, infix "@" (k: H): G is
        -- Entry of key `k'.
        require
            valid_key: valid_key (k)
        deferred
        end;

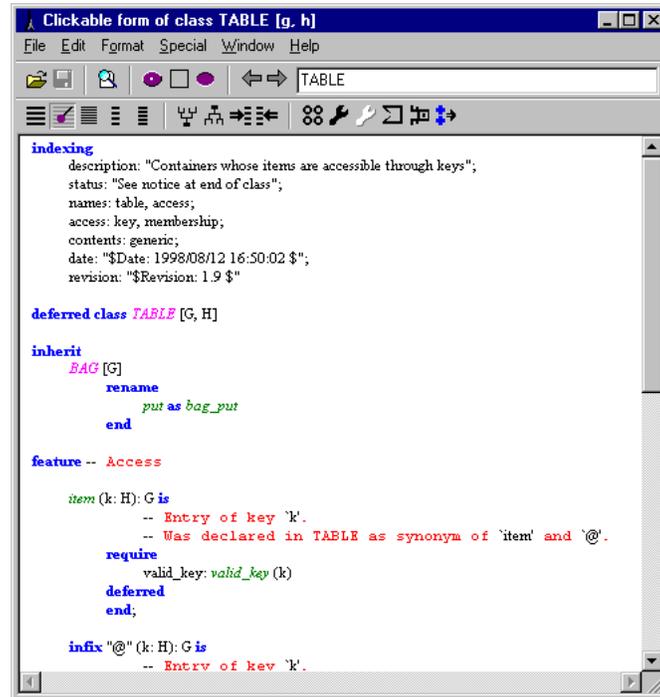
```

Since all class names in the following formats are clickable, you can hold down CTRL, and then right-click a class name to display its properties in a **Class Tool**.

To view a format below, click the corresponding button on the **Class** toolbar.

- **Text** : displays a textual representation of **TABLE** (default — see preceding illustration).

- **Clickable** : displays the active class with **keywords** in blue and bold, *identifiers* in italic, and **comments** in red; every semantically-meaningful component is clickable.



```

indexing
description: "Containers whose items are accessible through keys";
status: "See notice at end of class";
names: table, access;
access: key, membership;
contents: generic;
date: "$Date: 1998/08/12 16:50:02 $";
revision: "$Revision: 1.9 $"

deferred class TABLE [G, H]

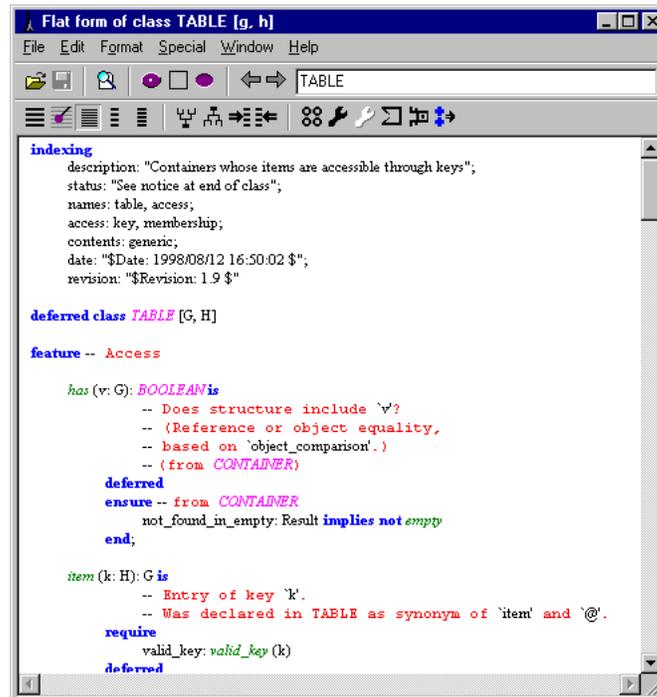
inherit
  BAG [G]
  rename
    put as bag_put
  end

feature -- Access

  item (k: H): G is
    -- Entry of key 'k'.
    -- Was declared in TABLE as synonym of 'item' and '@'.
    require
      valid_key: valid_key (k)
    deferred
    end;

  infix '@' (k: H): G is
    -- Entrv of key 'k'.
  
```

- **Flat** : provides the reconstructed form of **TABLE**, including all inherited features.



```

indexing
description: "Containers whose items are accessible through keys";
status: "See notice at end of class";
names: table, access;
access: key, membership;
contents: generic;
date: "$Date: 1998/08/12 16:50:02 $";
revision: "$Revision: 1.9 $"

deferred class TABLE [G, H]

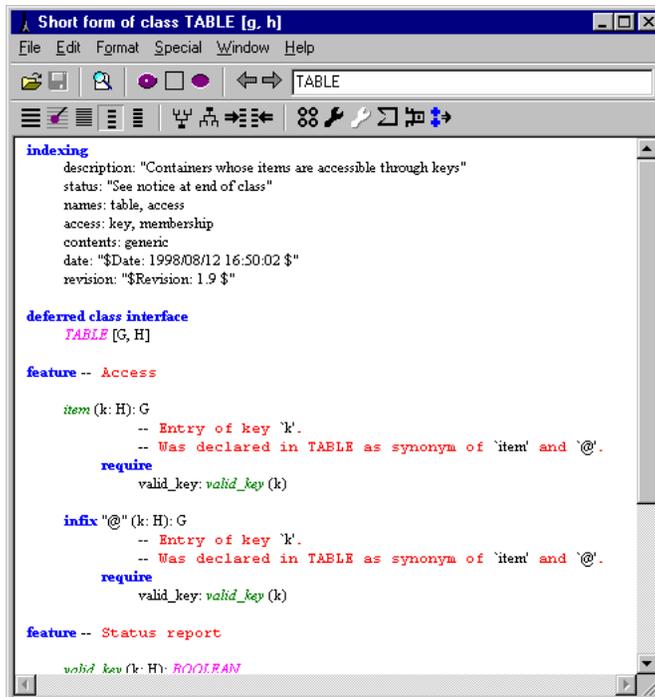
feature -- Access

  has (v: G): BOOLEAN is
    -- Does structure include 'v'?
    -- (Reference or object equality,
    -- based on 'object_comparison'.)
    -- (from CONTAINER)
  deferred
  ensure -- from CONTAINER
    not_found_in_empty: Result implies not empty
  end;

  item (k: H): G is
    -- Entry of key 'k'.
    -- Was declared in TABLE as synonym of 'item' and '@'.
    require
      valid_key: valid_key (k)
    deferred
  end;

```

- **Short** : displays the interface that **TABLE** offers its clients: it removes all implementation information (routine bodies, secret features), but keeps the routine headers, the header comments and assertions; this is the interface through which you will use any class.



```

Short form of class TABLE [g, h]
File Edit Format Special Window Help
TABLE
indexing
description: "Containers whose items are accessible through keys"
status: "See notice at end of class"
names: table, access
access: key, membership
contents: generic
date: "$Date: 1998/08/12 16:50:02 $"
revision: "$Revision: 1.9 $"

deferred class interface
TABLE [G, H]

feature -- Access

item (k: H): G
-- Entry of key `k`.
-- Was declared in TABLE as synonym of `item` and `@`.
require
valid_key: valid_key (k)

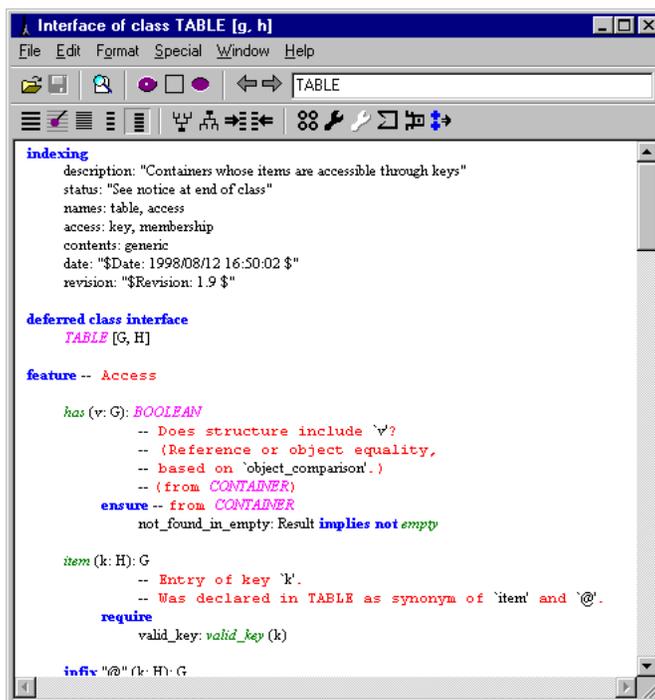
infix "@" (k: H): G
-- Entry of key `k`.
-- Was declared in TABLE as synonym of `item` and `@`.
require
valid_key: valid_key (k)

feature -- Status report

valid_key (k: H): BOOLEAN

```

- **Flat/short** : displays the short form of the flat — the complete interface.



```

Interface of class TABLE [g, h]
File Edit Format Special Window Help
TABLE
indexing
description: "Containers whose items are accessible through keys"
status: "See notice at end of class"
names: table, access
access: key, membership
contents: generic
date: "$Date: 1998/08/12 16:50:02 $"
revision: "$Revision: 1.9 $"

deferred class interface
TABLE [G, H]

feature -- Access

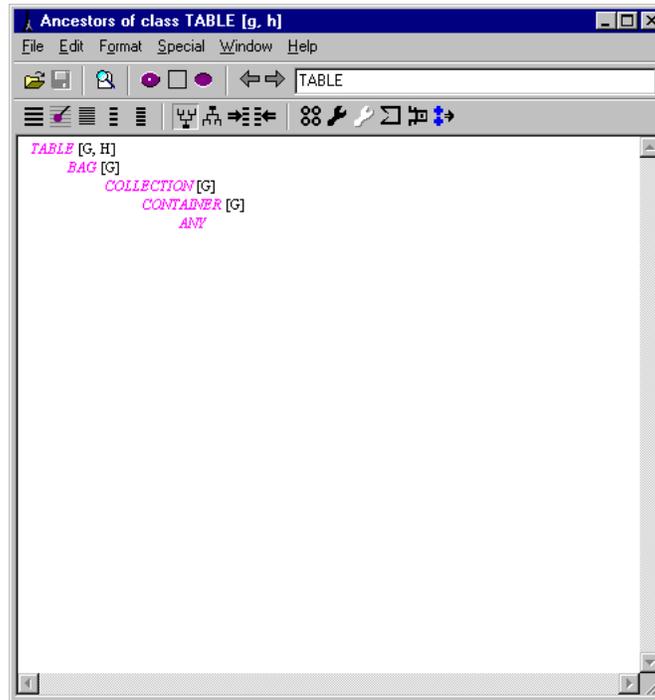
has (v: G): BOOLEAN
-- Does structure include `v`?
-- (Reference or object equality,
-- based on `object_comparison`.)
-- (from CONTAINER)
ensure -- from CONTAINER
not_found_in_empty: Result implies not empty

item (k: H): G
-- Entry of key `k`.
-- Was declared in TABLE as synonym of `item` and `@`.
require
valid_key: valid_key (k)

infix "@" (k: H): G

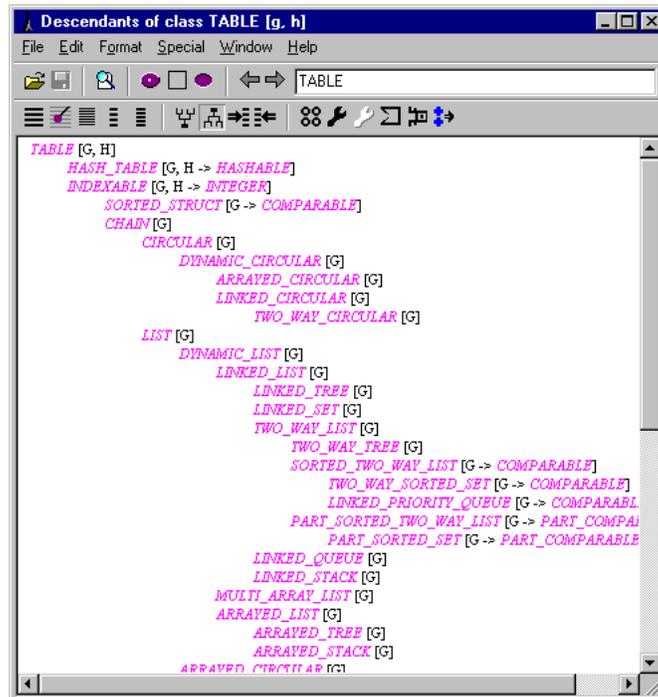
```

- **Ancestors** : lists the ancestor structure that leads to **TABLE**.

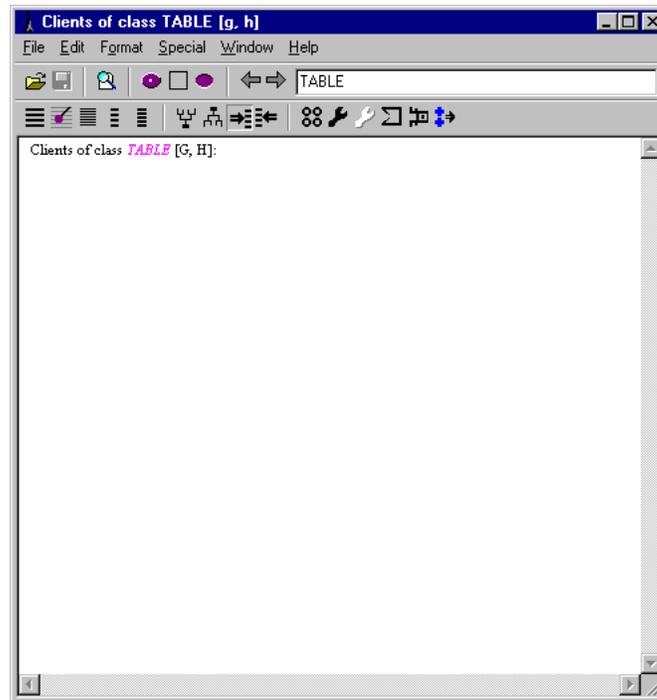


In the preceding, **TABLE** inherits from **BAG**, which inherits from **COLLECTION**, and so on, while the inheritance structure stops at **ANY** — an ancestor of all developer-written classes.

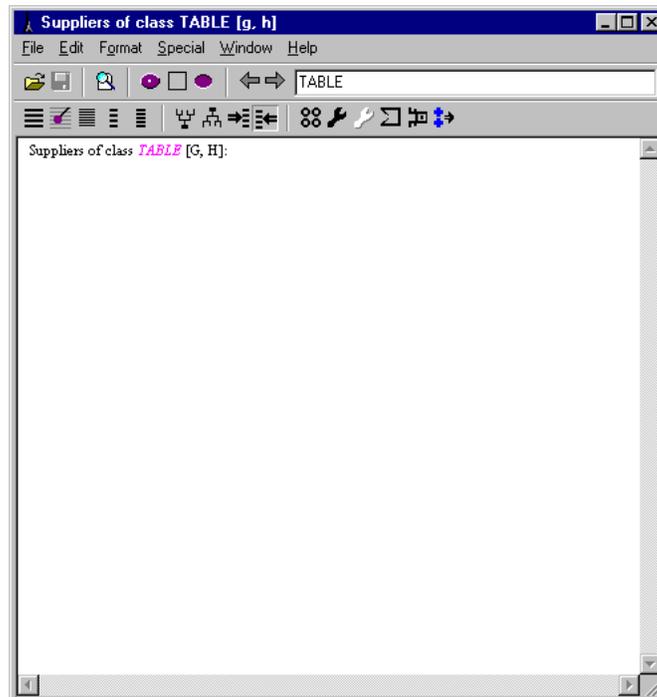
- **Descendants** : lists all descendants of **TABLE**.



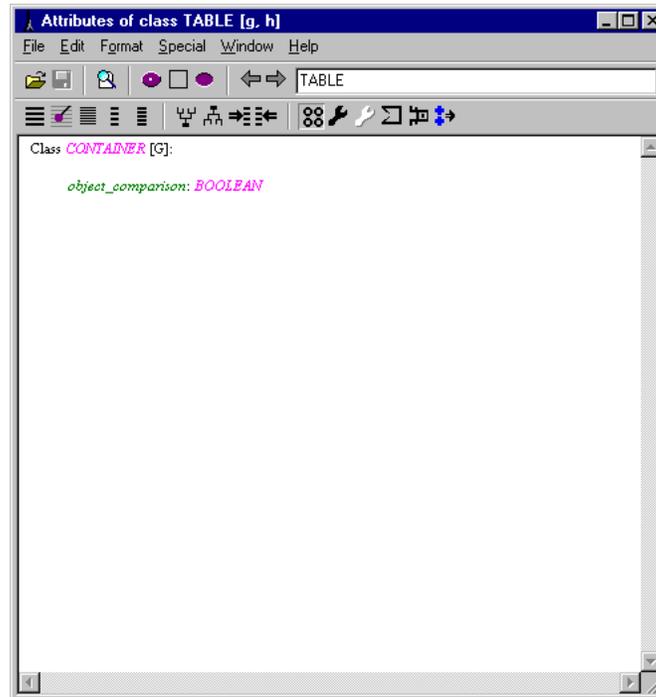
- **Clients**  : lists all clients of **TABLE**.



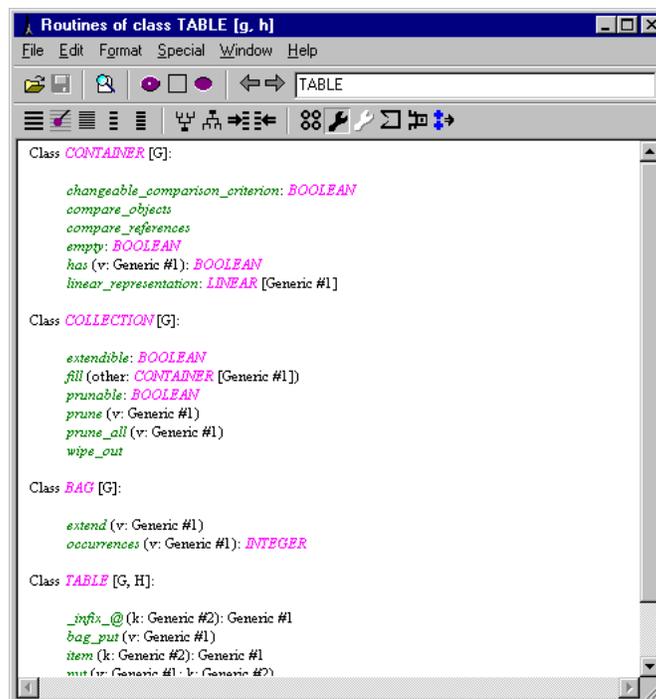
- **Suppliers**  : lists all suppliers of **TABLE**.



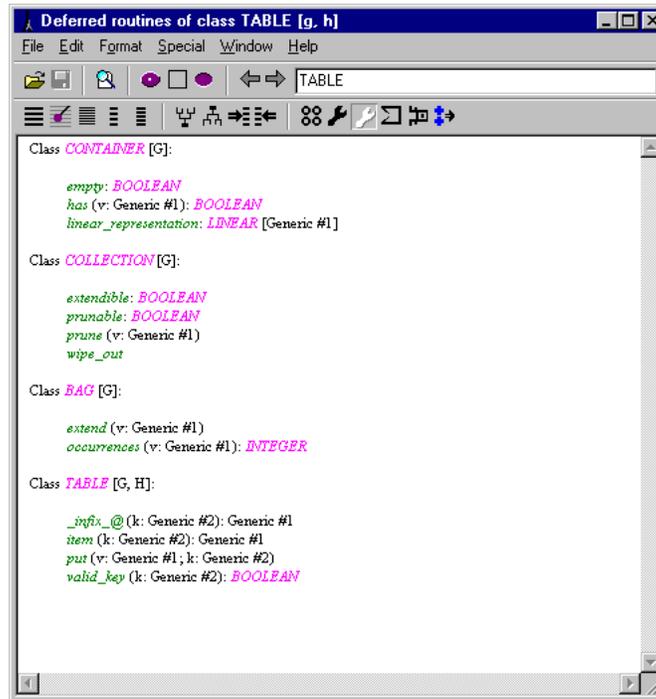
- **Attributes** : lists all attributes of **TABLE**, sorted by originating class.



- **Routines** : displays information about all routines in **TABLE**, sorted by originating class.



- **Deferred**  : lists all deferred routines in **TABLE**, sorted by originating class.

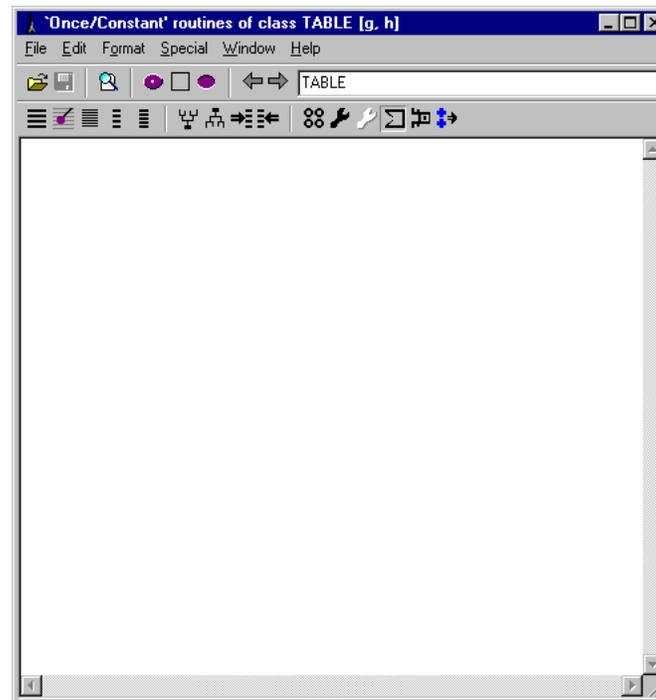


```

Deferred routines of class TABLE [g, h]
File Edit Format Special Window Help
TABLE
Class CONTAINER [G]:
  empty: BOOLEAN
  has (v: Generic #1): BOOLEAN
  linear_representation: LINEAR [Generic #1]
Class COLLECTION [G]:
  extendible: BOOLEAN
  prunable: BOOLEAN
  prune (v: Generic #1)
  wipe_out
Class BAG [G]:
  extend (v: Generic #1)
  occurrences (v: Generic #1): INTEGER
Class TABLE [G, H]:
  _infix_@ (k: Generic #2): Generic #1
  item (k: Generic #2): Generic #1
  put (v: Generic #1, k: Generic #2)
  valid_key (k: Generic #2): BOOLEAN

```

- **Once/Constants**  : lists all once routines and constants in **TABLE**, sorted by originating class.

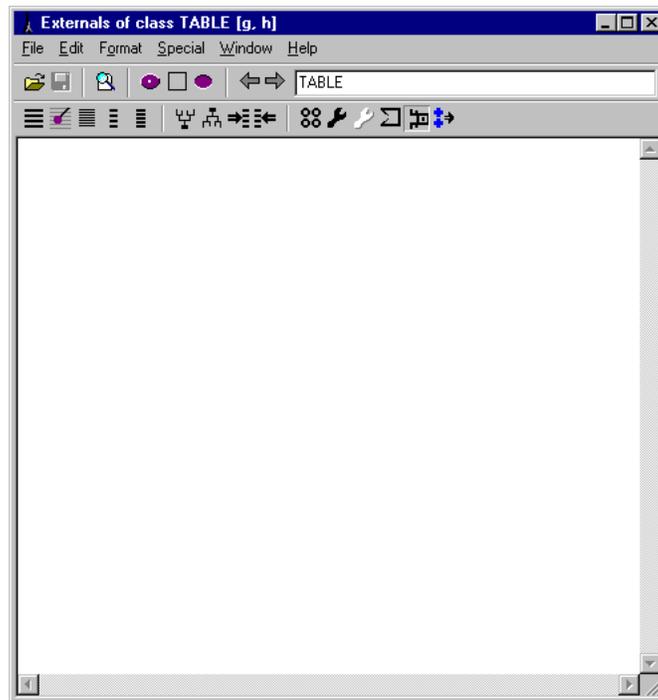


```

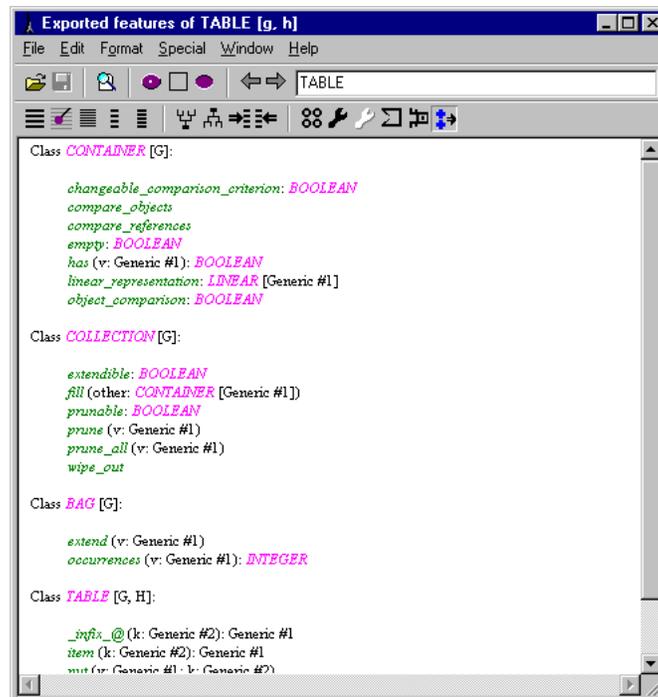
Once/Constant' routines of class TABLE [g, h]
File Edit Format Special Window Help
TABLE

```

- **Externals** : lists all external routines in **TABLE**, sorted by originating class.



- **Exported** : lists all exported routines in **TABLE**, sorted by originating class.



Note the first routine defined in **TABLE**, **_infix_@**. In the class text, it displays as **infix “@”**; this is the Eiffel mechanism for routines used in infix forms, such as **infix “+”** on integers.

In this representation, it means that a client can access the entry of key **k** in table **t** through the notation **t @ k** as an alternative to **t.item (k)**. In a list of features (like the preceding one), the conventional syntax is **infix_x**, where **x** represents the infix operator.

Note the use of the synonym mechanism of Eiffel: a feature has two names, an identifier name (**item**) and an infix name (**infix "@**"). The effect is the same as if you had two declarations, identical except for the feature name.

In the next section, you will explore the contents of several features using the **Feature Tool**. For the sake of continuity, you will start with feature **extend** (from class **TABLE**). You must first retarget the **Feature Tool** to **extend**.

To target the **Feature Tool** to **extend**:

- 1 In the **Feature Tool**, in the left **Target Name Tool**, type **extend**, and then press TAB.
- 2 In the right **Target Name Tool**, type **TABLE**, and then press ENTER.

Feature Tool formats

The features of a class include both the inherited and immediate features. There are two possible types of features:

- **attribute** — contains information stored with each instance of the class and returns a result. Attributes cannot have arguments.
- **routine** — describes an algorithm applicable to every instance of the class. If the routine returns a result, it is called a **function**; if not, it is called a **procedure**.

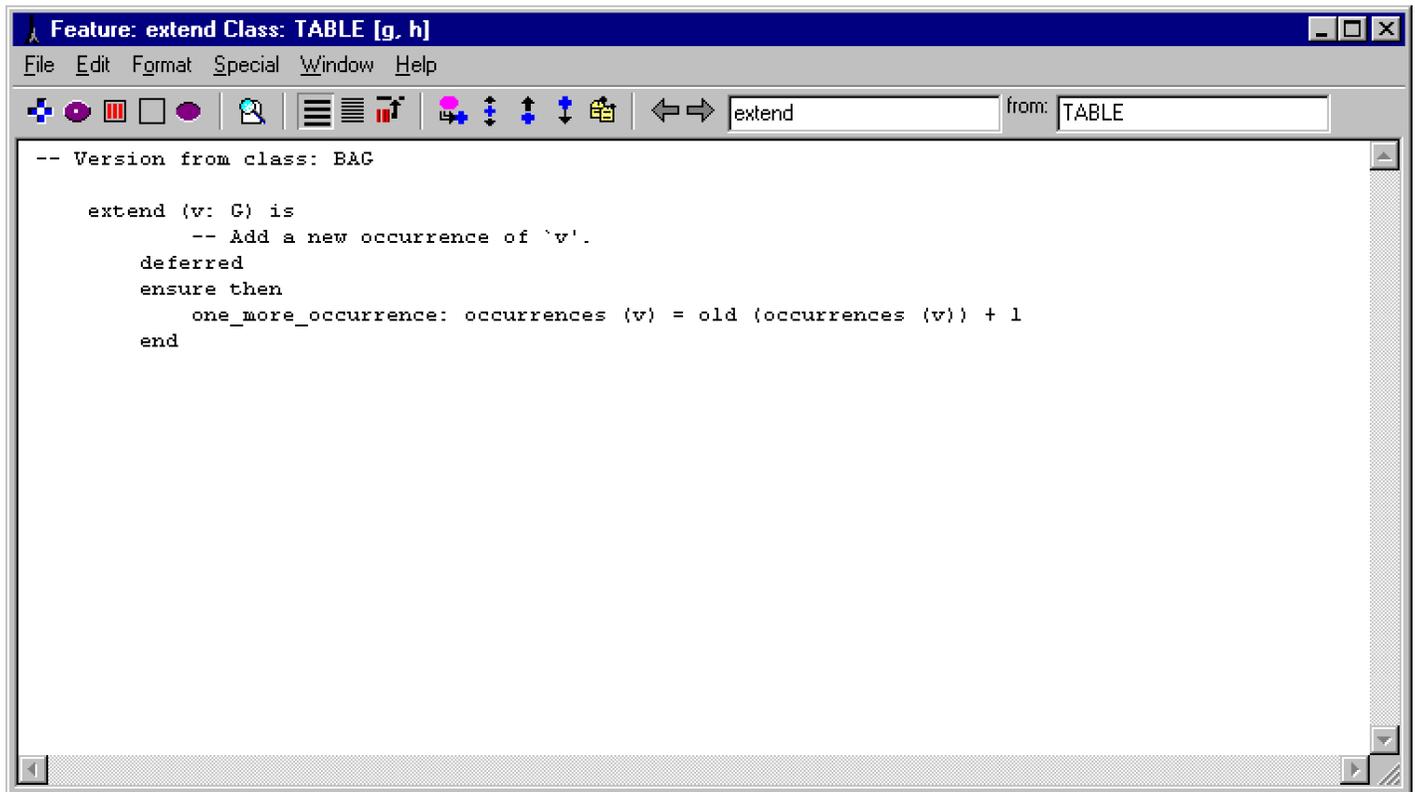
This classification is based on the implementation technique of the feature, as seen from the class itself. Viewed from the perspective of a client class, a feature can also be classified as either:

- a **command** — the feature can change its target object, but does not return a result. A command may only be implemented as a *procedure*.
- a **query** — the feature returns a result, but should not change the object. A query may be implemented as either a *function* or an *attribute*.

and its implementation status in the class is one of two kinds:

- **effective** — the feature is completely implemented.
- **deferred** — the feature is *specified* in the current class, but not implemented yet; implementations may be provided in proper descendants.

Like the two previous tools, the **Feature Tool** appears displaying the text for the active feature (**extend**) in the tool window. You can use the commands on the **Feature** toolbar to view the contents of **extend** in different formats.

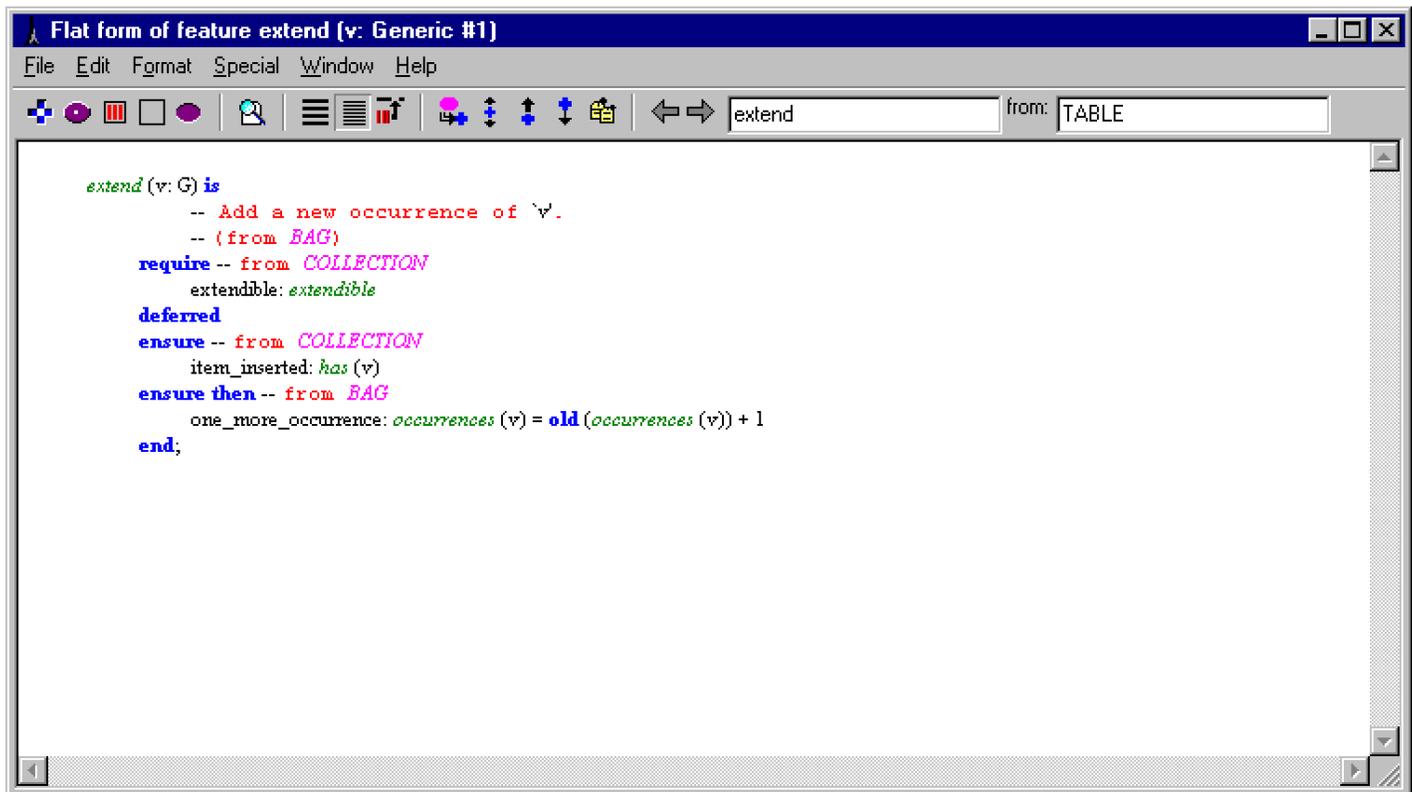


Since many of the objects in the following formats are clickable, you can hold down CTRL, and then right-click an object to display its properties.

To view a format below, click the corresponding button on the **Feature** toolbar.

- **Text** : displays a textual representation of **extend** (default — see preceding illustration).

- **Flat** : lists the reconstructed version of **extend**, with all inherited assertions combined.



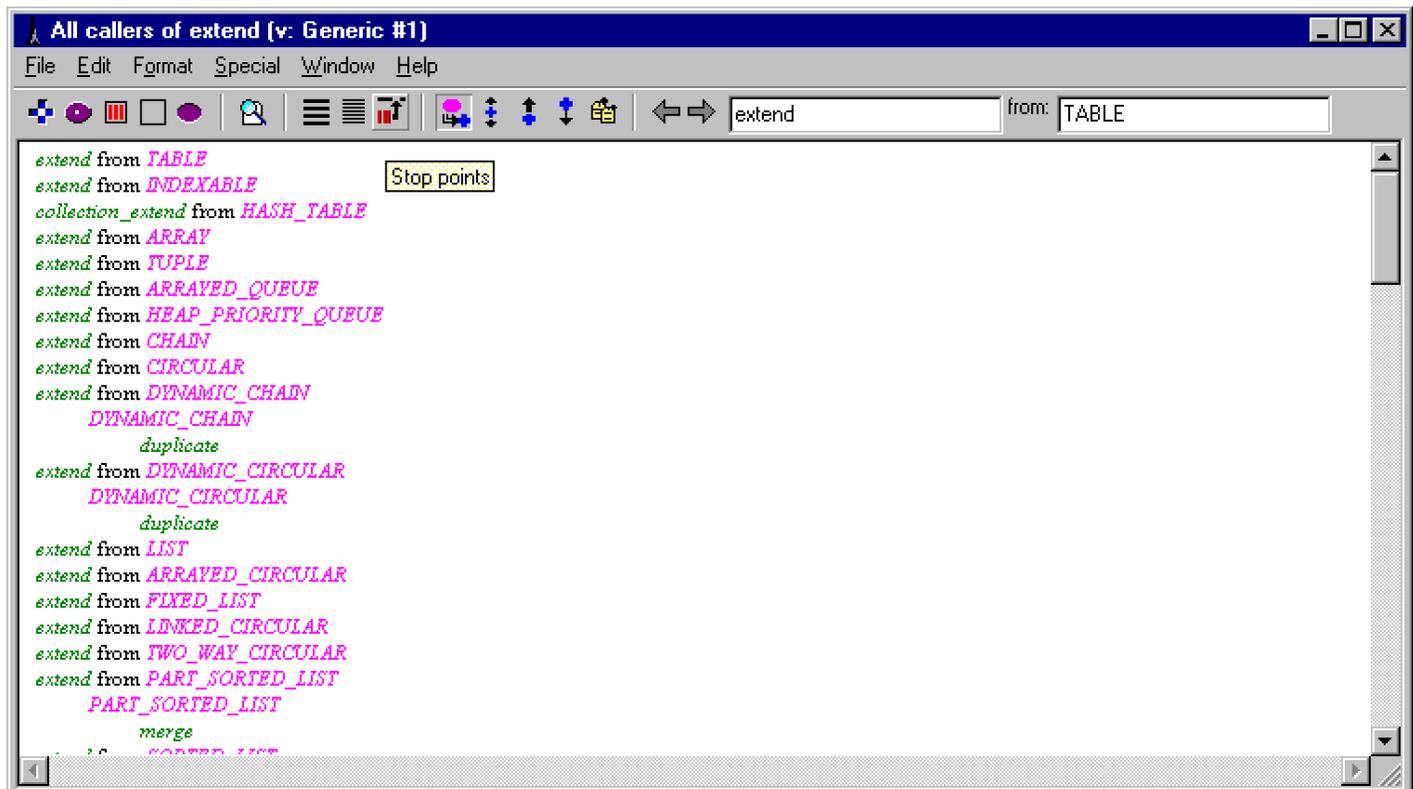
```

Flat form of feature extend (v: Generic #1)
File Edit Format Special Window Help
extend from: TABLE

extend (v: G) is
  -- Add a new occurrence of v.
  -- (from BAG)
  require -- from COLLECTION
    extendible: extendible
  deferred
  ensure -- from COLLECTION
    item_inserted: has (v)
  ensure then -- from BAG
    one_more_occurrence: occurrences (v) = old (occurrences (v)) + 1
end;

```

- **All callers** : lists all locations in the system where the feature is called.



```

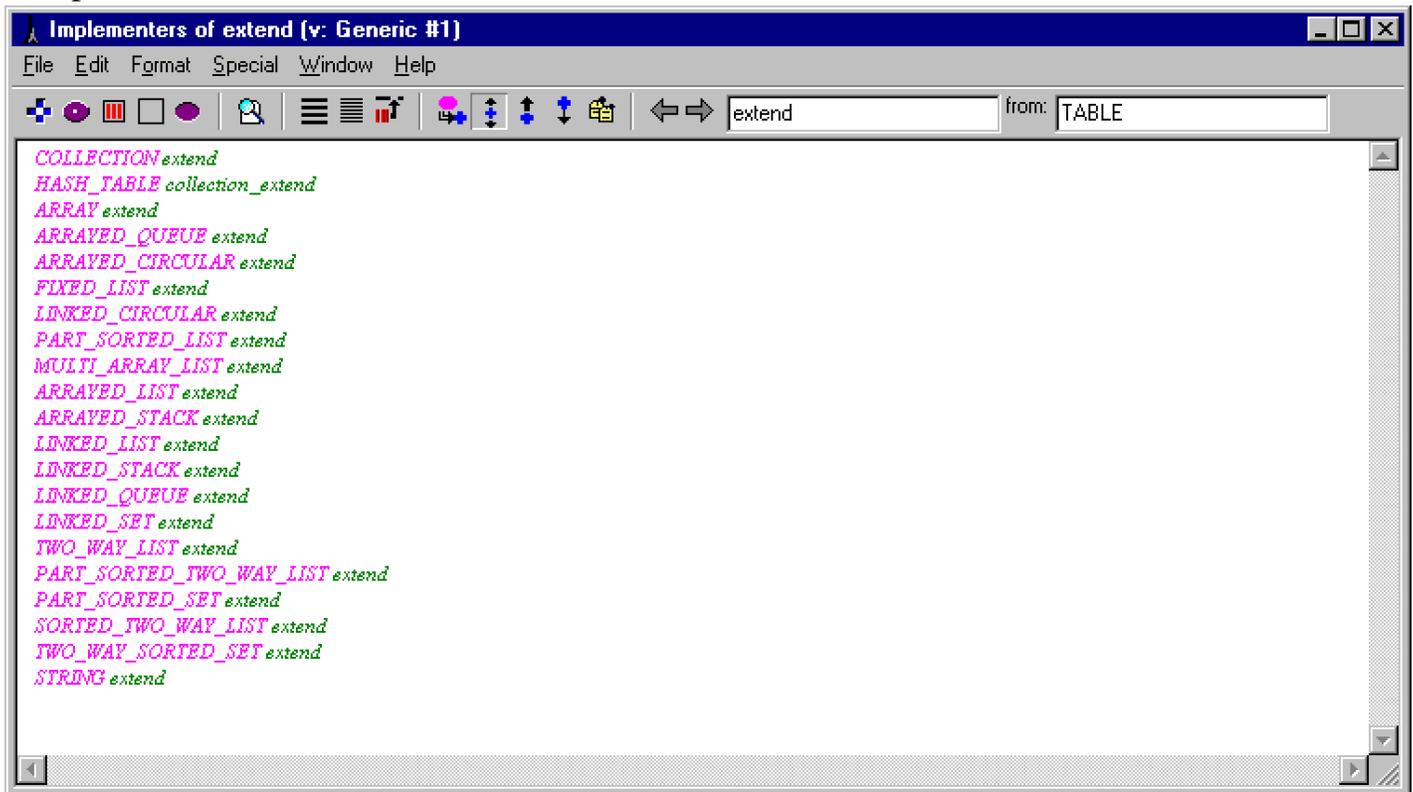
All callers of extend (v: Generic #1)
File Edit Format Special Window Help
extend from: TABLE

extend from TABLE
extend from INDEXABLE
collection_extend from HASH_TABLE
extend from ARRAY
extend from TUPLE
extend from ARRAYED_QUEUE
extend from HEAP_PRIORITY_QUEUE
extend from CHAIN
extend from CIRCULAR
extend from DYNAMIC_CHAIN
DYNAMIC_CHAIN
duplicate
extend from DYNAMIC_CIRCULAR
DYNAMIC_CIRCULAR
duplicate
extend from LIST
extend from ARRAYED_CIRCULAR
extend from FIXED_LIST
extend from LINKED_CIRCULAR
extend from TWO_WAY_CIRCULAR
extend from PART_SORTED_LIST
PART_SORTED_LIST
merge
PART_SORTED_LIST

```

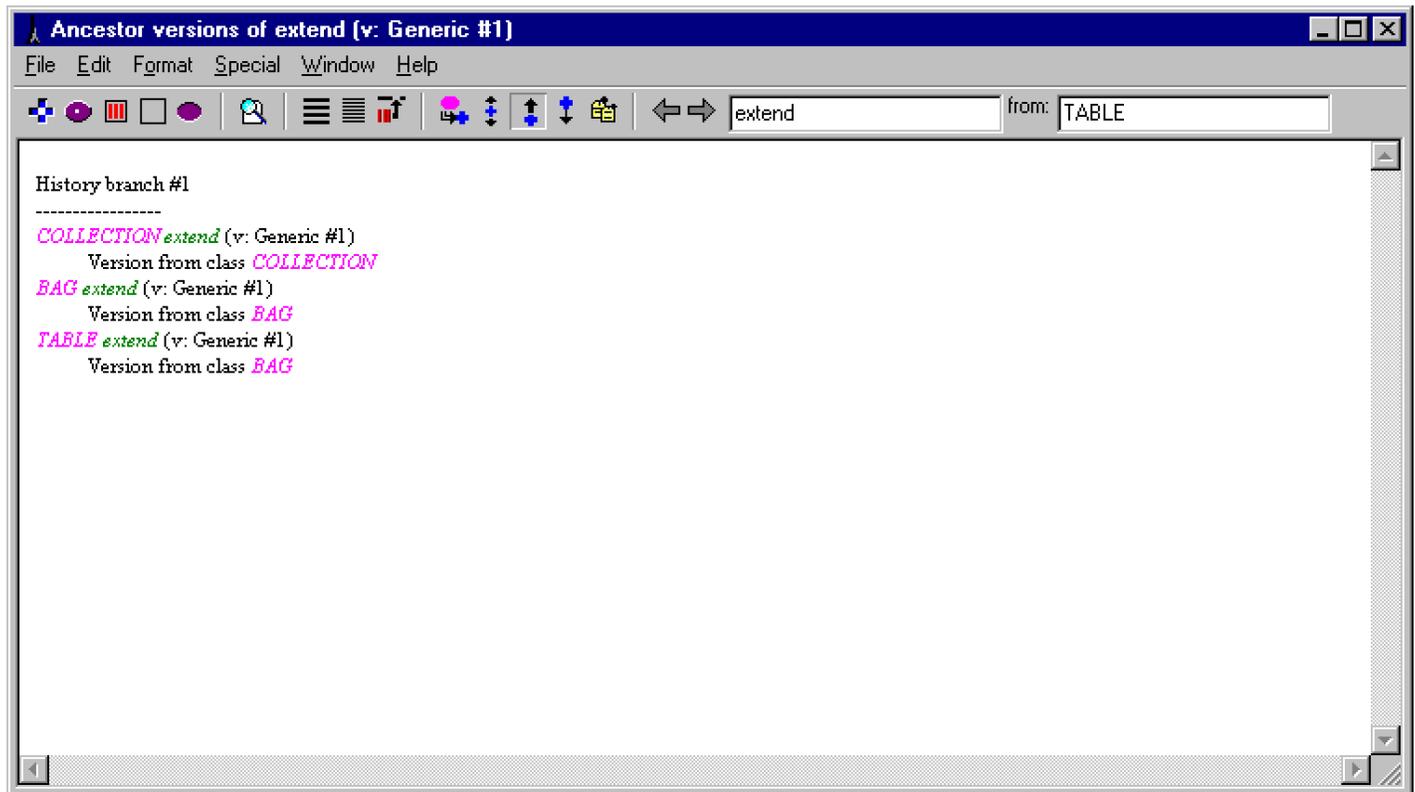
As you can see, the **Feature Tool** title bar displays the signature of the routine as **extend (v: GENERIC #1)**. This means that **extend** takes one argument, whose type is the first (and only) generic parameter of the class, called **G** within the class.

- **Implementers** : lists all redeclared versions of **extend**. Redeclaration retains the original name of a feature, but changes other properties, such as the implementation.



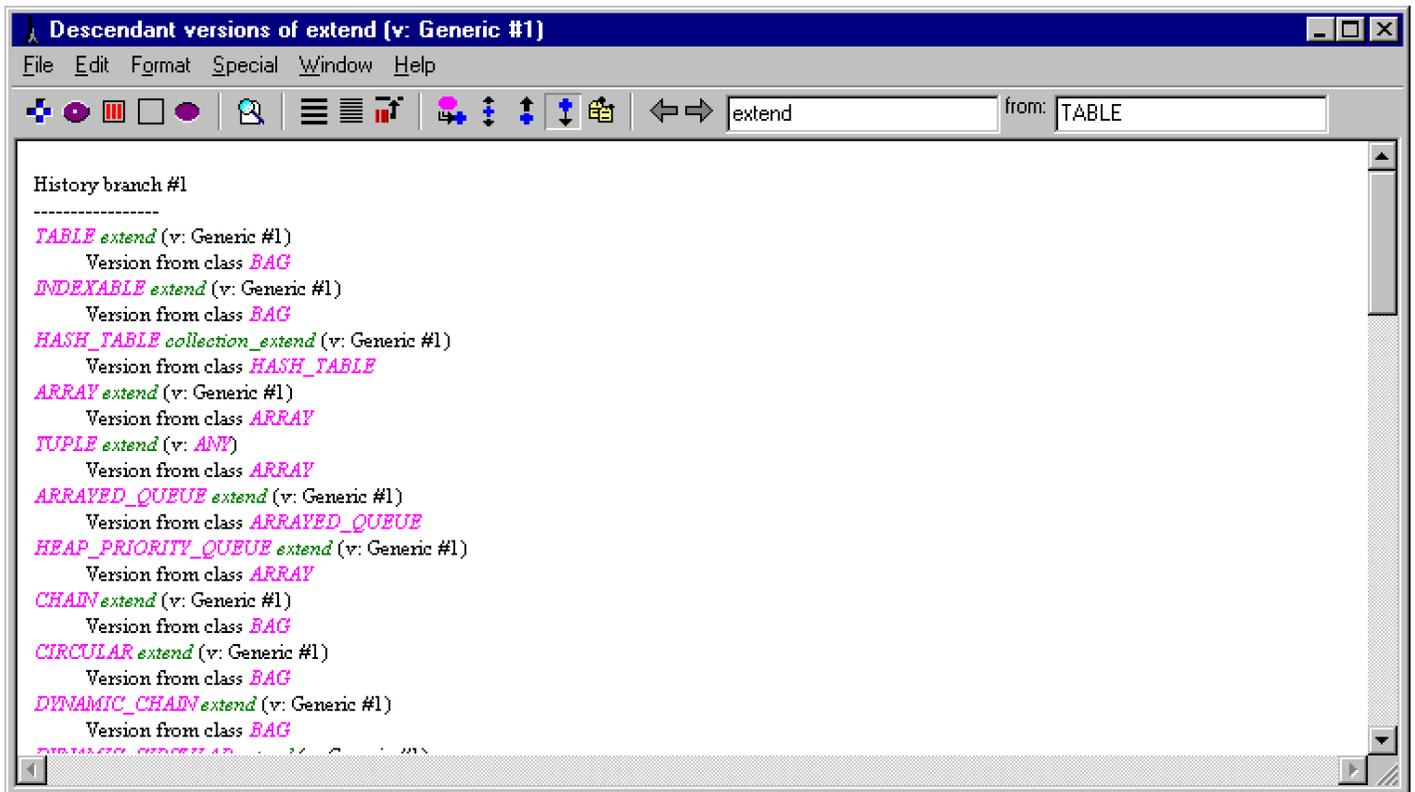
The preceding illustration shows all classes where **extend** from **LIST**, which originally comes from **COLLECTION**, has a different, redeclared version.

- **Ancestor versions** : displays the entire history of **extend**: its versions in every ancestor; not just those in which **extend** is redeclared.

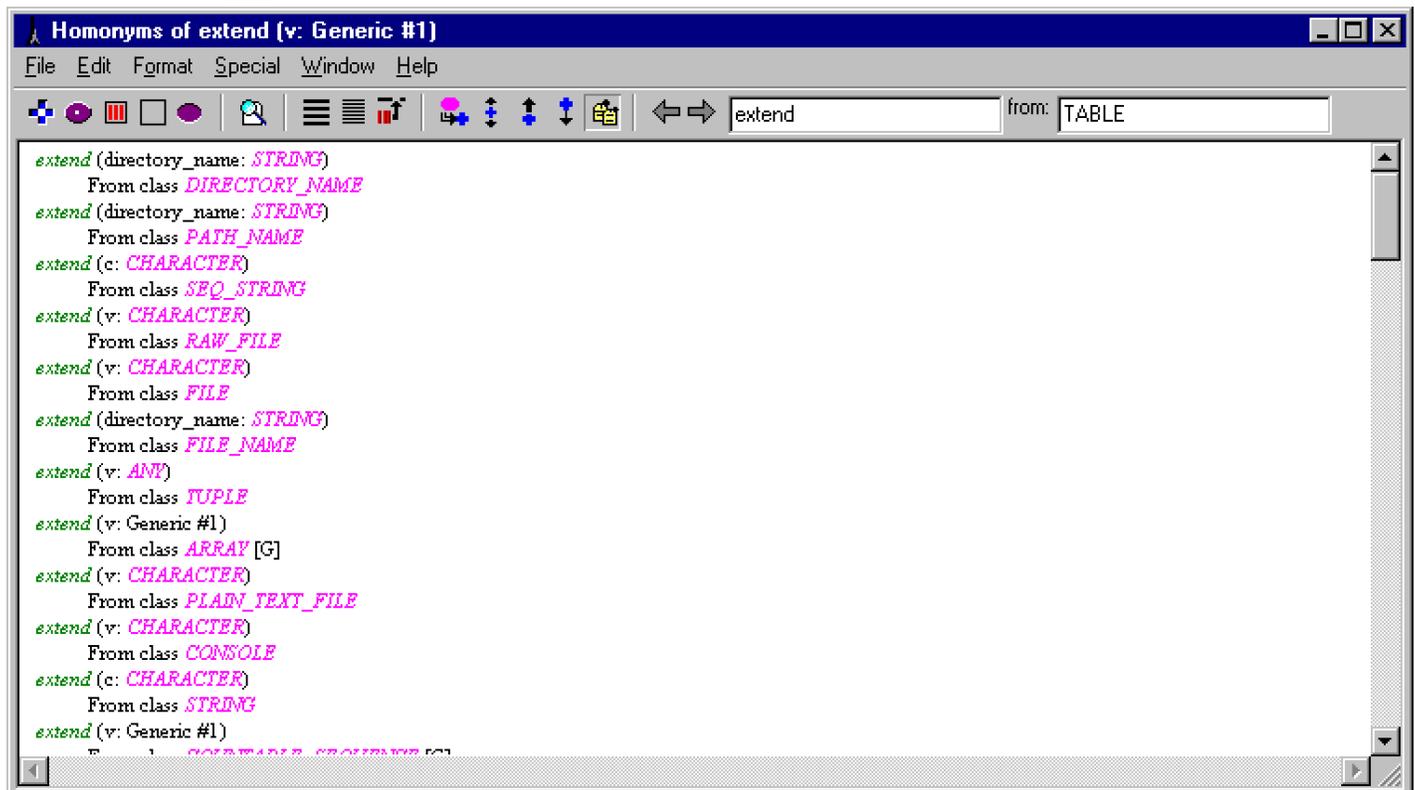


You will normally use the pick-and-drop operation with this format to see what an ancestor version looks like. In this example, there is only one ancestor (**History branch #1**), but in multiple inheritance cases that involve joining features from several ancestors, there will be several.

- **Descendant versions**  : lists all descendants of **extend**.



- **Homonyms**  : lists all features called **extend** in the active system.



Unlike the features which display in the preceding formats, these features are not necessarily related to **extend** from **TABLE**: they simply have the same name. Since

EiffelBench must explore the entire system representation, rather than rely on structural information, it takes awhile to display this format.

The large number of features named **extend** in many different classes are the result of the rules for systematic feature naming for libraries in the Eiffel method. For more information on these rules, see *Reusable Software: The Base Object-Oriented Component Libraries*.

If you have experience with object-oriented software construction, the importance of the facilities explored to this point become apparent. Specifically, given the power and versatility of polymorphism and feature redeclaration (redefinition and effecting), it is essential to quickly answer the following questions:

- What does feature **f** become in class **C**?
- What are the descendents of **C**?
- From what class does the version of **f** in **C** come?
- What was the original version of **f**?

The techniques described in the preceding sections answer the most common version of these questions.

In the next section, you will modify class **HEIR**, and then recompile using **Quick Melt**  on the **Project Tool** toolbar.

1.12 Modifying and remelting

To continue this tutorial, use one of the methods discussed earlier in this chapter to retarget a **Class Tool** to **HEIR**, and then select **Text**  as the display format.

In the following section, you will add procedure **new_message** after procedure **display** and before **display_routine**, and then add a call to **new_message** at the beginning of **display**.

Modifying HEIR

To modify **HEIR**:

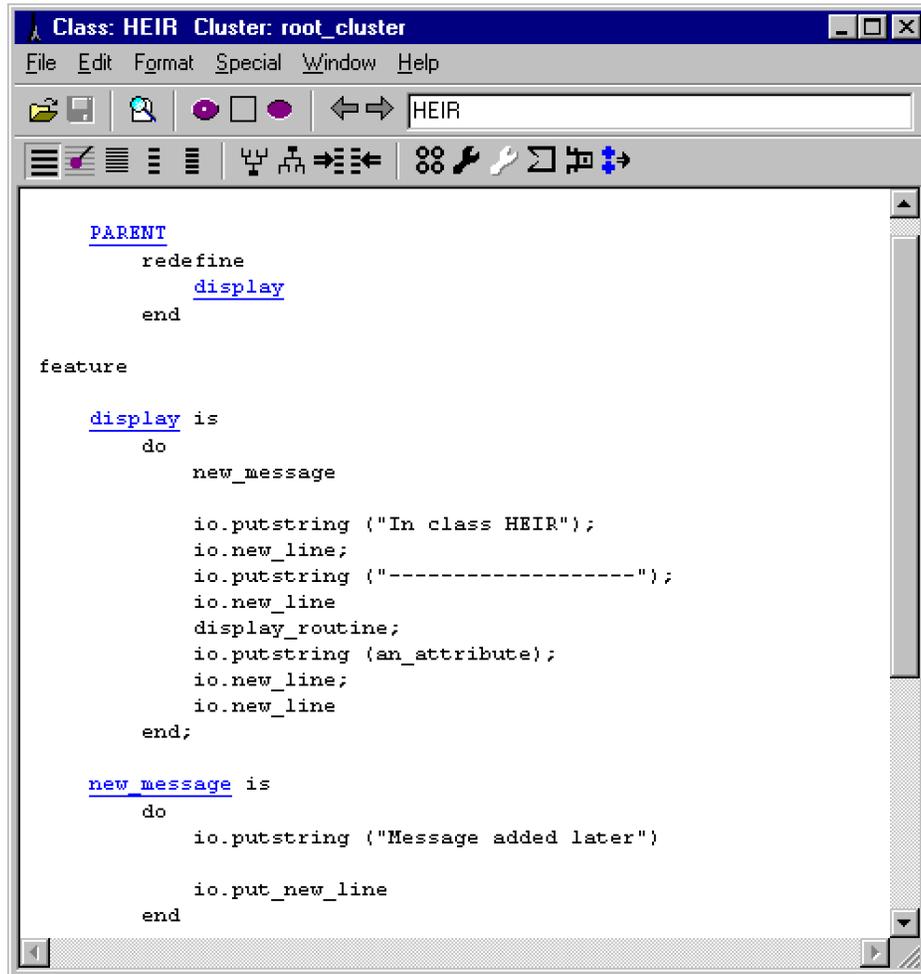
- 1 In the **Class Tool** window, point to the space above **display_routine is**, and then click.
- 2 Press ENTER, and then type the following:

```
new_message is
  do
    io.put_string ("Message added later")

    io.put_new_line
  end
```

- 3 Point to the space below **do** (below **display is**), and then click.

- 4 Press ENTER, and then type `new_message`.
- 5 On the **Class Tool** toolbar, click **Save** .



The screenshot shows a window titled "Class: HEIR Cluster: root_cluster". The window contains the following Eiffel code:

```

PARENT
  redefine
    display
  end

feature

  display is
  do
    new_message

    io.putstring ("In class HEIR");
    io.new_line;
    io.putstring ("-----");
    io.new_line
    display_routine;
    io.putstring (an_attribute);
    io.new_line;
    io.new_line
  end;

  new_message is
  do
    io.putstring ("Message added later")

    io.put_new_line
  end

```

Having saved the changes to **HEIR**, you can now recompile the project.

Recompiling HEIR

To recompile **HEIR**:

- On the **Project Tool** toolbar, click **Quick Melt** .

During the melting process, a message of the form “**Degree *n*: class XXX**” appears in the **Project Tool**. Since EiffelBench automatically determines the smallest possible set of classes to recompile, fewer message display than during the initial compilation.

If you encounter a compilation error and the correction is not obvious, read through the next two sections, which discusses errors and error correction.

1.13 Syntax errors

When the compiler detects an error in either the Eiffel code or the Ace file, a message appears in the workspace. Some elements of the message, such as class names or

error codes, are clickable — you can use the pick-and-drop operation to drop them on the appropriate hole to get more detailed information.

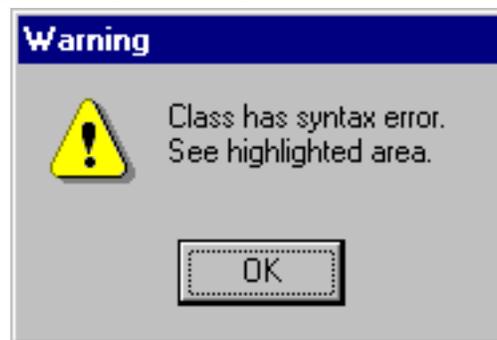
To view this mechanism, you will introduce a syntax error.

Creating a syntax error

To create a syntax error:

- 1 In the **Class Tool** window (targeted to **HEIR**), point to **display is** (below **feature**).
- 2 Replace **is** with **ist**, and then on the **Class Tool** toolbar, click **Save** .

Since EiffelBench parses the class as it saves, you do not need to click **Quick Melt**. The following error message dialog box displays:



It is a general principle of ISE Eiffel that, whenever possible, message dialog boxes are **non-preemptive** — instead of having the program stop responding until you click OK, you can continue working as the dialog box displays. Once you correct the error, the dialog box closes.

Correcting the syntax error

To correct the syntax error:

- 1 In the **Class Tool** window, point to **ist**, and then click.
- 2 Backspace over **t**, and then on the **Class Tool** toolbar, click **Save** .

1.14 Validity errors

The only errors of substance in Eiffel are those that cause violations of one of the validity constraints defined in the language definition. A four letter code of the form **VXXX** (the first letter is always **V**) identifies each constraint. A validity violation produces a precise error message, which includes the validity code.

Although short, you can usually determine what the error is from its message. If not, you can use the pick-and-drop operation to drop the error message on **Explanation**  on the **Project Tool** toolbar to display the complete text of the violated constraint.

Like the syntax error section, you will introduce a validity error to view this mechanism. In this section, you must first use one of the methods introduced earlier in this chapter to retarget the **Class Tool** to **TESTROOT**.

```

class TESTROOT creation

    make

feature

    o1, o2: PARENT;

    make is
    do
        display_demonstration_message
        !HEIR! o1
        !! o2
        o1.display
        o2.display
    end

    display_demonstration_message is
    do
        io.put_new_line
        io.put_string (" ISE Eiffel spoken here")
        io.put_new_line
        io.put_string ("-----*N*N*");
    end

    -- To get a typical compilation error, remove the two dashes
    -- at the beginning of the next line:
    -- inv: INVALID

```

At the end of class **TESTROOT**, directly above the final **end**, there is the following comment line:

```
-- inv: INVALID
```

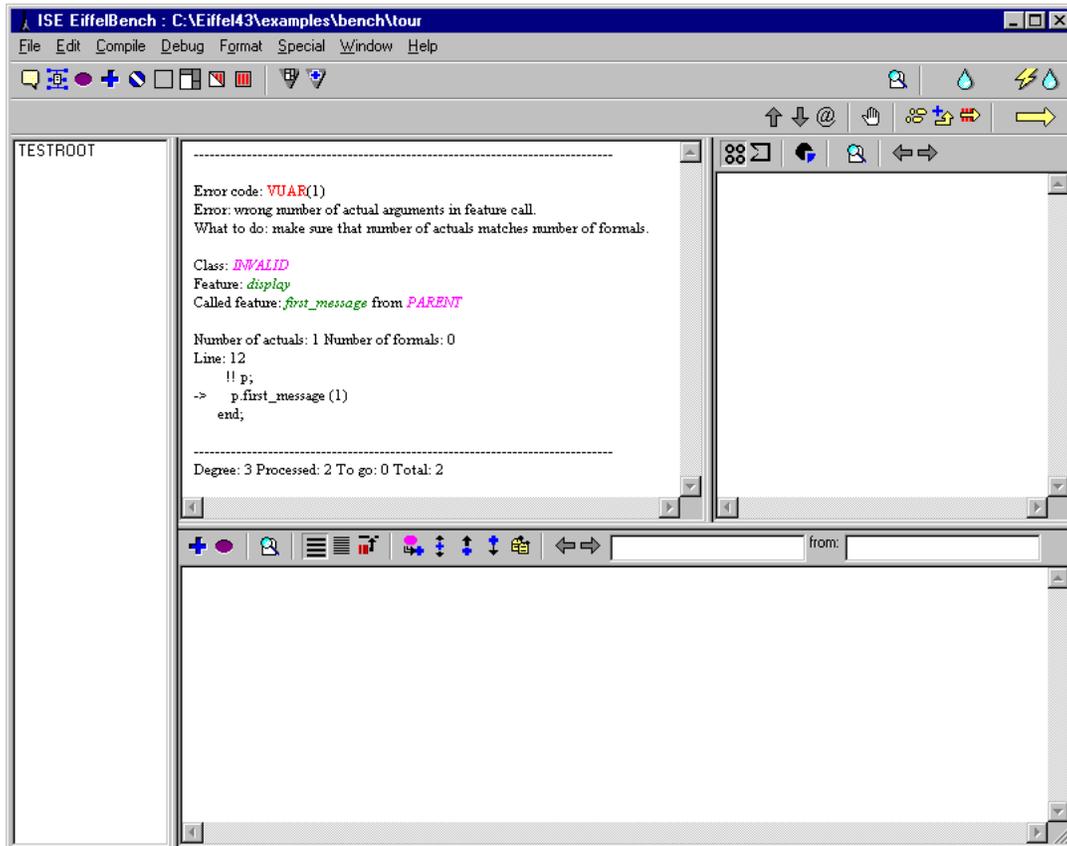
Uncommented, this is a declaration of a feature of type **INVALID**. A class called **INVALID** exists in file `invalid.e` of the root cluster, which contains a validity error.

Creating a validity error

To create a validity error:

- 1 In the **Class Tool** window (targeted to **TESTROOT**), point to **--inv: INVALID**, and then click.
- 2 Backspace over **--**, on the **File** menu, click **Save**, and then click **Quick Melt** .

The compiler executes degrees 5, 4, and 3 on **TESTROOT** and **INVALID** before the following displays:



Here again, class and feature names in this message are clickable, which makes it easy to find out what went wrong: procedure **display** of class **INVALID** calls **first_message** from **PARENT** with one argument, but the procedure does not take any arguments.

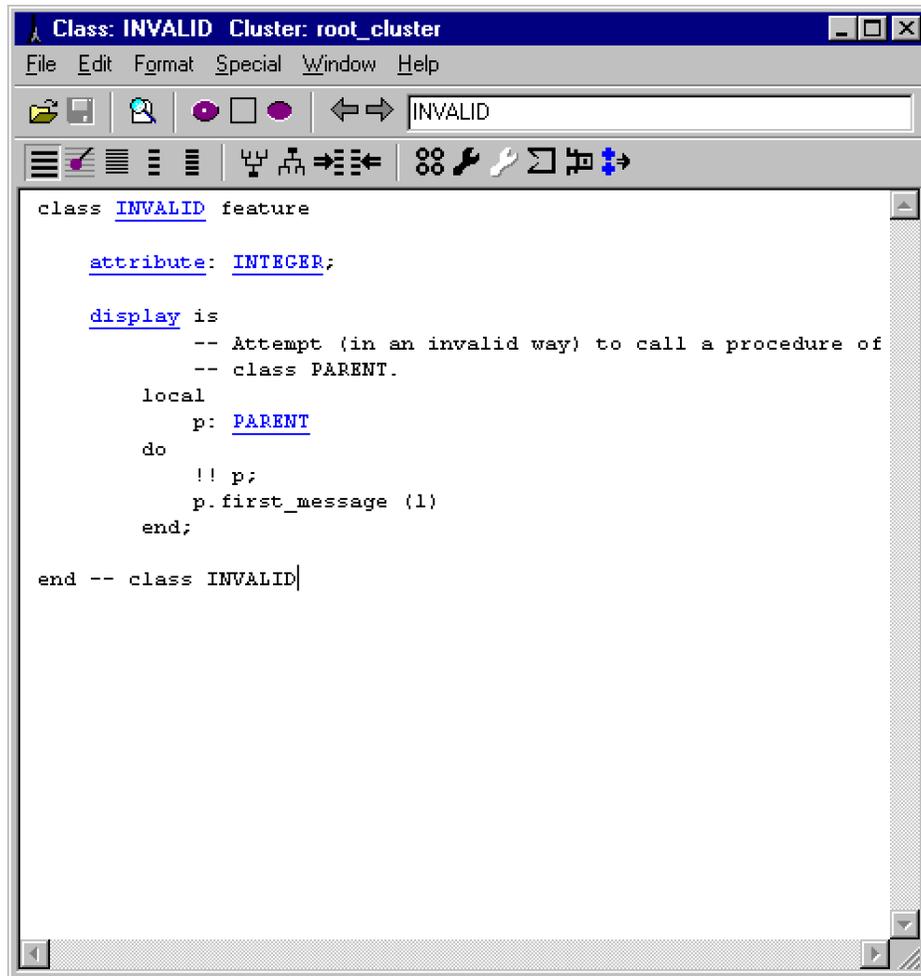
The error code **VUAR** is also clickable. You can use the pick-and-drop operation to drop the error code on the **Explanation** hole in the **Project Tool** to display the complete text of the violated rule from *Eiffel: The Language*.

Since the error code in the Project Tool reads **VUAR(1)**, this means that the first clause is the one violated. This convention of displaying the clause number in parentheses applies to all multi-clause validity constraints.

Correcting the validity error

To correct the validity error:

- 1 In the **Project Tool** window, use the pick-and-drop operation to drop **display** on the **Class Tool** window (targeted to **TESTROOT**).



- 2 In the **Class Tool** window, point to **p.first_message(1)**, and then backspace over **(1)**.
- 3 On the **Class Tool** toolbar, click **Save** , and then click **Quick Melt** .

In the next section you will finally run the system.

1.15 Running the system

To run the system:

- On the **Project Tool** toolbar click **Run** .

On Windows, a MS-DOS-based program window displays the output. On Unix, Linux or VMS you can easily display the output. Since you start EiffelBench from a specific window when you type **ebench**, that window is where the standard output

goes and from where the execution reads standard input. You can use the standard operating system mechanisms to redirect the standard input and output.

To continue the tutorial, you need to close the MS-DOS-based program window (Windows only).

Closing the MS-DOS-based program window

To close the MS-DOS-based program window:

- Press ENTER.

On all platforms, you can rerun the system as often as you want — just click **Run** again.

1.16 Arguments

Since the system used in this tutorial is extremely simple, execution arguments are not necessary. In more complex systems, you may want to pass values to the execution, such as a numeric parameter or a file name, so that you can have different executions without changing and recompiling the software.

In Eiffel text, you can access run-time arguments using a feature of the Kernel Library class **ARGUMENTS**. Any class in a system can inherit from **ARGUMENTS** and can use the following queries:

- **argument_count** — returns the number of arguments passed to the execution.
- **argument (i)** — returns a string representation for the *i*-th argument.
- **argument_i** — returns the string representation for the *i*-th element.

You can retarget a **Class Tool** to **Arguments** to display a complete list of all its features — on the **Class Tool** toolbar, click **Short** .

The following section outlines the steps required to define execution arguments. Since the active system does not require any arguments, the information is solely for reference.

Defining execution arguments

To define execution arguments in EiffelBench:

- 1 On the **Project Tool** toolbar, right-click **Run** .



- 2 In the **Specify arguments** box, type all arguments, separated by a space.
- 3 Do one of the following:
 - To start an execution that uses the arguments, click **Run**.
 - To save the arguments for later use, click **OK**.

To continue the tutorial, click **Cancel**.

1.17 Debugging

Before completing the tutorial, it is important that you explore the debugging facilities, which allow you to watch what occurs during execution — on the control side (successive instructions) and to explore development objects.

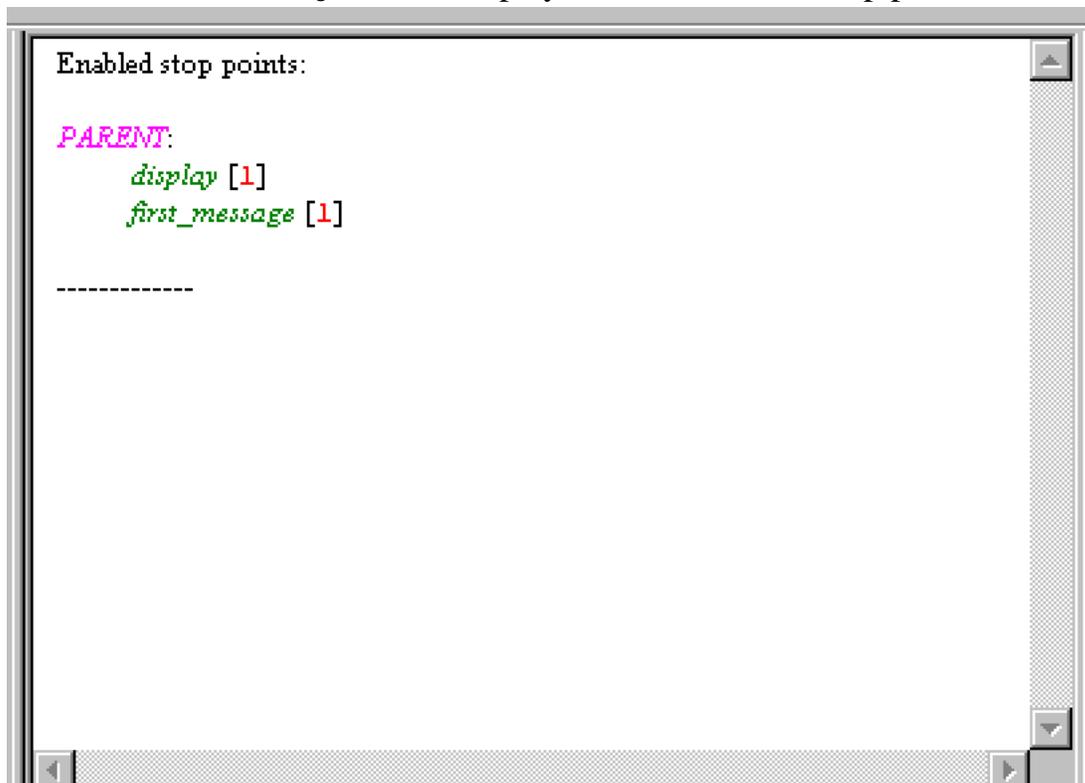
Located on the **Project Tool** toolbar, **Stop points**  controls the execution of a routine. Before you continue, use one of the methods discussed earlier in this chapter to retarget a **Class Tool** to **PARENT**.

Adding stop points to a routine

To add stop points to a routine:

- 1 Use the pick-and-drop operation to drop **display** on the **Stop point** hole ICON on the **Project Tool** toolbar.
- 2 Repeat for **first_message**.

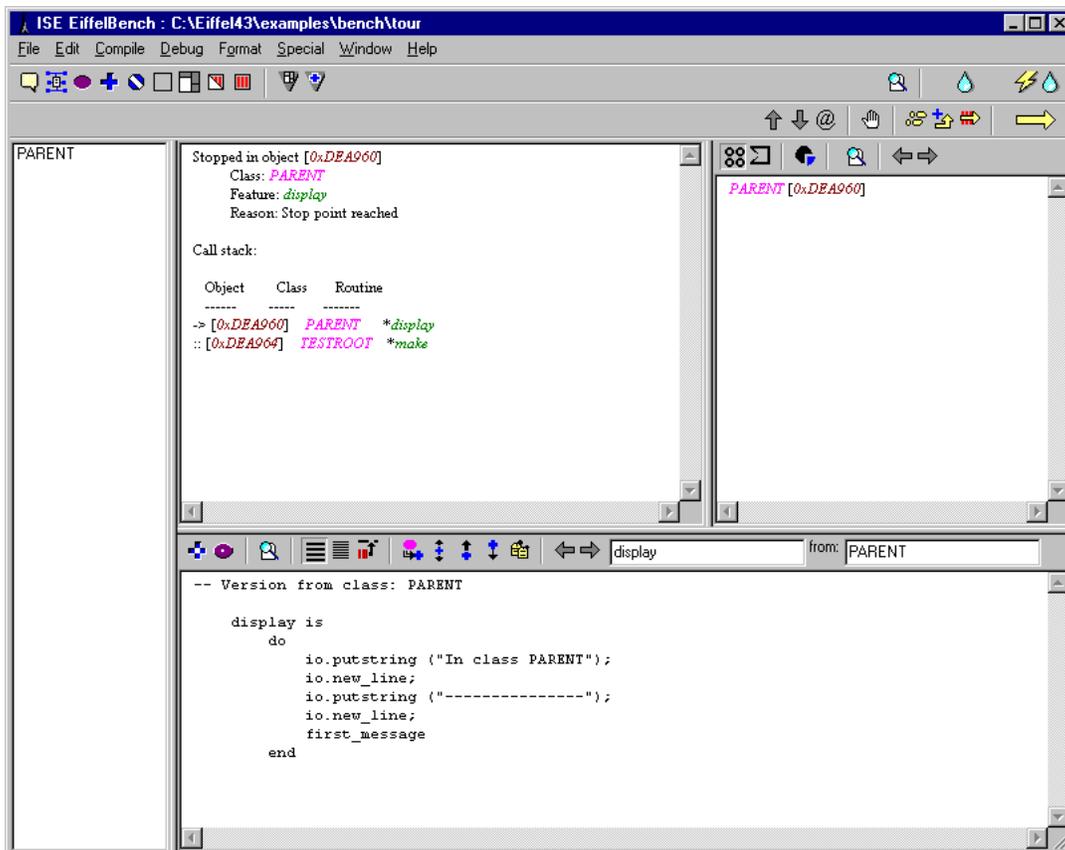
The **Output window** in the **Project Tool** displays the list of active stop points:



The number in brackets [] following the routine name indicates the position of the stop point in the routine. In this example, each stop point is at the beginning of the routine ([1]).

3 On the **Project Tool** toolbar, click **Run** .

The program stops at the first stop point and the **Project Tool** displays the following information:



- **Stop point reached** — reason for interrupting the execution.
- **Call stack** — shows the state of the execution (from bottom to top):

The first call applied procedure **make** to the root object — an instance of class **TESTROOT** identified as **[0xDEA964]**. This triggered a call to procedure **display** of class **PARENT** applied to object **[0xDEA960]**. If you look at the text of **TESTROOT**, you will see that this is the object attached to attribute **o2** and is a results of the second creation instruction in the procedure **make**.

The preceding also introduces the last major type of development abstraction — **execution objects** — represented by the object identifiers, such as **[0xDEA960]**. Execution objects are run-time objects created during execution as instances of classes in the system.

The numbers that identify execution objects are internal codes and determine whether two execution objects are the same. Consequently, the numbers that display when you run this example will differ from the preceding illustration.

Since the numbers represent class instances, you can hold down CTRL, and then right-click to view the corresponding run-time object and its fields. For example, you can hold down CTRL, and then right-click **display** to display the body of the routine where execution stopped.

Removing a stop Point

To remove a stop point:

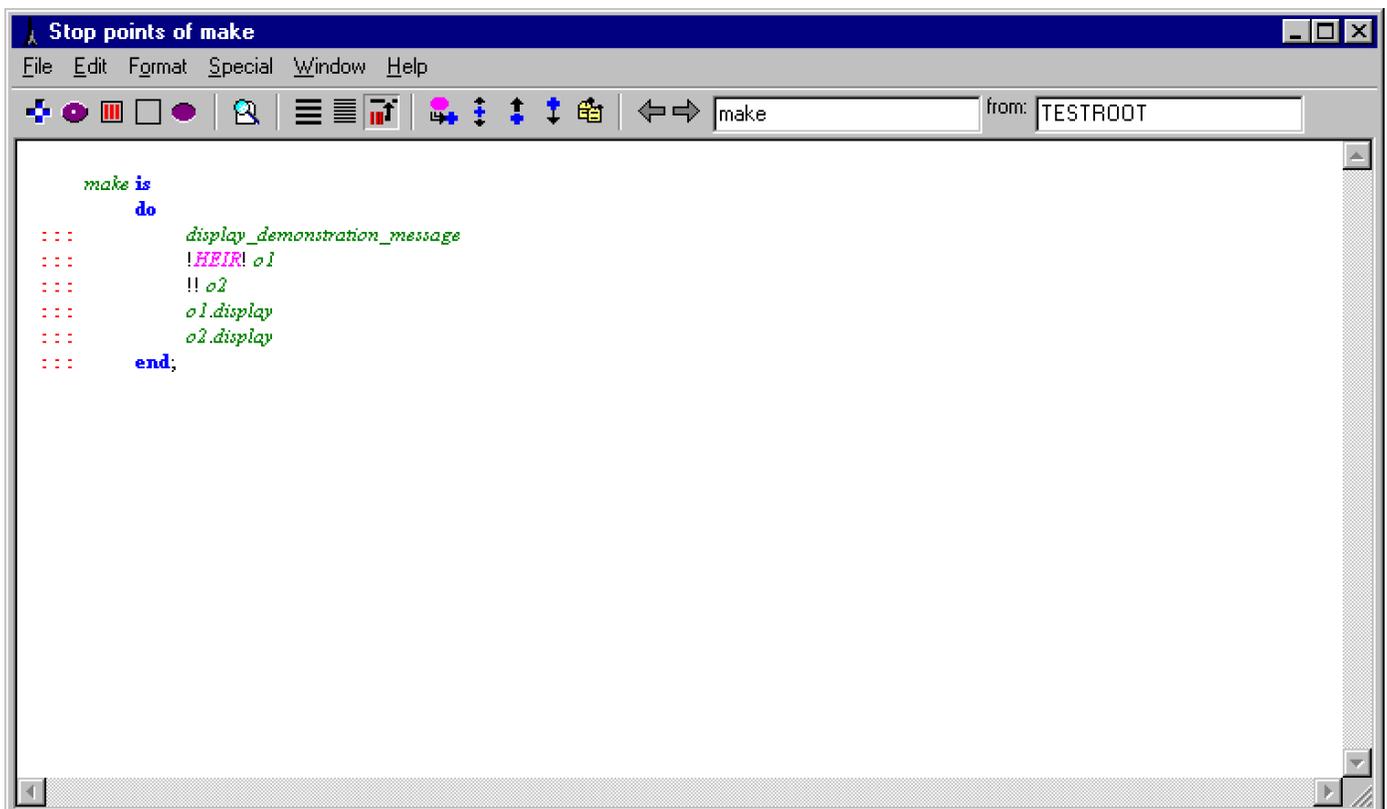
- Use the pick-and-drop operation to drop a stopped routine on **Stop point**  on the **Project Tool** toolbar.

Setting Stop points using the Feature Tool

The **Feature Tool** window in the **Project Tool** displays the **Stop point**  format. During execution, you can apply this format to any **Feature Tool**.

To set **Stop points** using the **Feature Tool**:

- 1 In the **Output window** in the **Project Tool**, hold down CTRL, and then right-click **make**.
A new **Feature Tool** appears, targeted to **make**.
- 2 On the **Feature Tool** toolbar, click **Stop points** .



The red markers in the preceding correspond to potential stop points, one for each instruction in the routine. There are three types of markers:

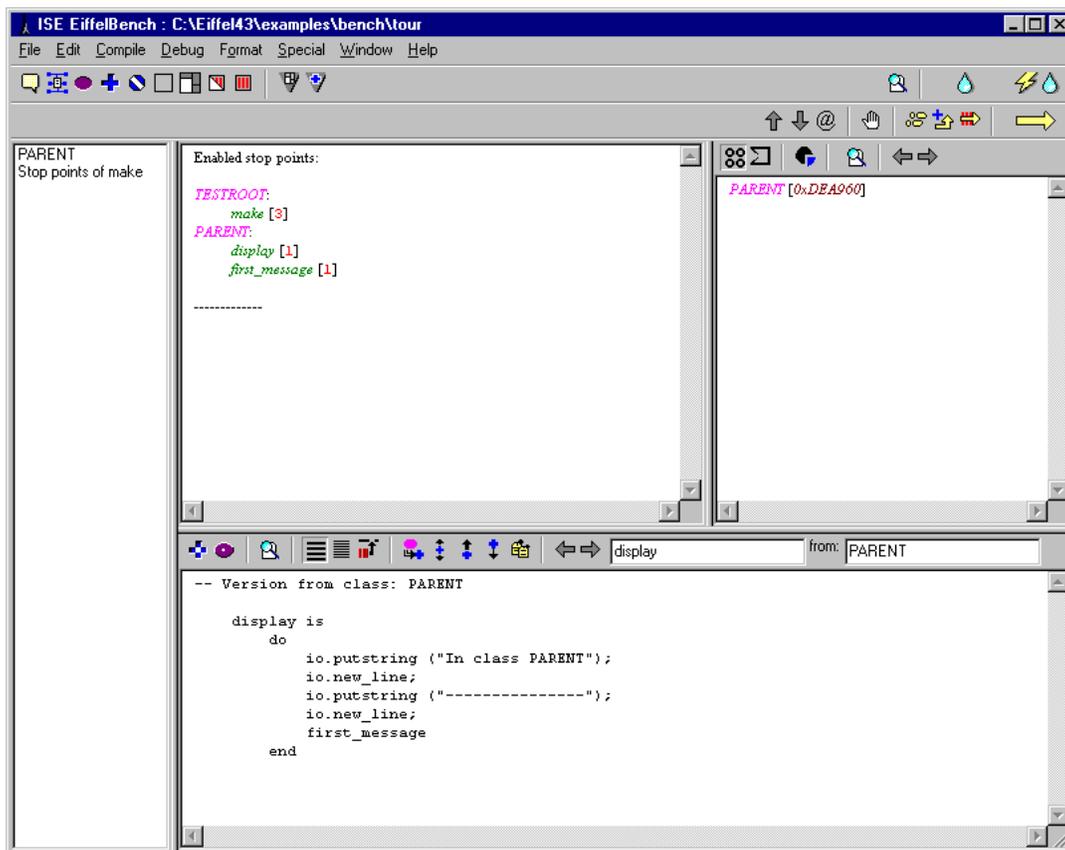
- **⋮** — a potential stop point.
- **|||** — an actual stop point.
- **-->** — the active instruction/execution position.

You can use the pick-and-drop operation to drop a marker on a **Stop point** hole to add a stop point to an individual instruction (in this case, **!!o2**).

Adding a stop point to an individual instruction

To add a stop point to an individual instruction:

- In the **Feature Tool**, use the pick and drop operation to drop the marker to the right of **!!o2** (**⋮**) on **Stop points** .



The **Output window** in the **Project Tool** displays the new stop point as **make [3]**, where **3** represents the third instruction. Since this instruction was already passed, the new stop point is useless in this execution. However, it applies to subsequent executions.

The display in the **Output window** also switches to **Stop point** format, a view that shows all active stop points. To redisplay the current state of execution, on the **Project Tool** toolbar, click **Execution status** .

To return to the **Stop point** format, on the **Project Tool** toolbar, click **Stop points**



1.18 Execution formats and stoppable routines

To resume execution, click **Run** . The first few lines of output appear in either the MS-DOS-based program window (Windows) or the shell window (Unix, Linux or VMS). Execution stops at the next stop point — the one at the beginning of procedure **first_message** in class **PARENT** — and the **Output window** redisplay the **Execution status** format.

The **Project Tool** toolbar also contains a series of execution formats that define how to execute, but do not execute until you do one of the following:

- Click **Run**  — restarts execution.
- Hold down CTRL and click the corresponding format button — defines a format and executes immediately.

The preceding is an example of another general convention of ISE Eiffel: in many cases, when you hold down CTRL and click a button, the command associated with the button occurs along with something complimentary. For example, when you hold down CTRL and click **Quick Melt** , the program recompiles and then executes.

To view a format below, click the corresponding button on the **Project Tool** toolbar.

- **Run** : executes until a stop point or exception is reached or the program terminates.
- **Step-by-step** : executes instruction by instruction in the applicable routines.
- **Out of routine** : executes the current routine to completion.
- **Ignore stop points** : ignores all stop points.
- **Clear stop points** : deletes all stop points in the program. You can also click **Clear stop points** on the **Debug** menu in the **Project Tool** to delete all stop points.

Disabling stop points

To disable stop points:

- Use the pick-and-drop operation to drop a stop point on a **Stop point** hole.

Unlike the **Clear stop points** command, the disabled stop point remains in the display. You can use the pick-and-drop operation a second time to enable the stop point.

To continue the tutorial, on the **Project Tool** toolbar, click **Step-by-step** .

The execution pointer `-->` is now next to the first instruction of the routine: `io.put_string ("In class PARENT")`. To move the execution pointer to the next instruction, click **Step-by-step** .

Stopping execution

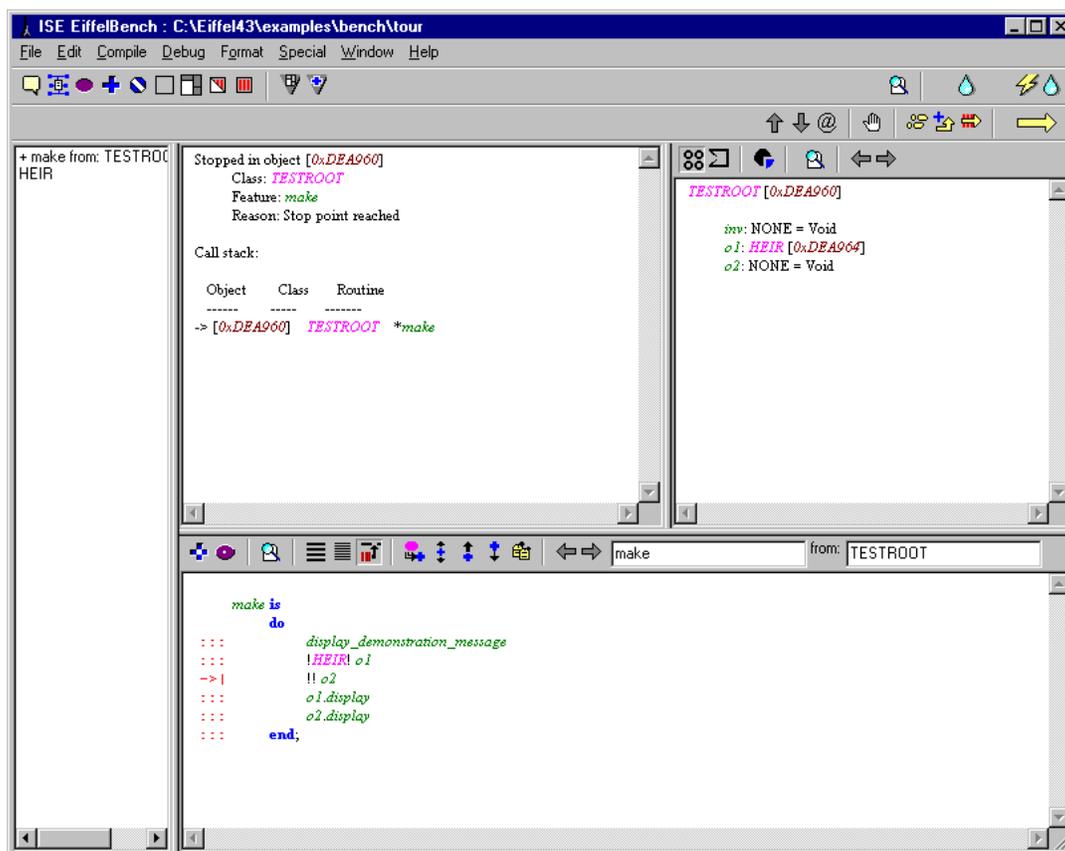
To interrupt a system as it executes, click **Interrupt**  on the **Project Tool** toolbar. This stops execution in the closest **stoppable routine**. A stoppable routine is one to which you added at least one stop point — even if you have since disabled or removed the stop point.

Since the active system is currently stopped, to terminate execution, hold down CTRL and click **Interrupt** . **System terminated** displays in the **Output window** in the **Project Tool**.

To continue the tutorial, click **Run** . Since you added a stop point to the third instruction of the creation procedure `make` (in class `TESTROOT`), execution stops.

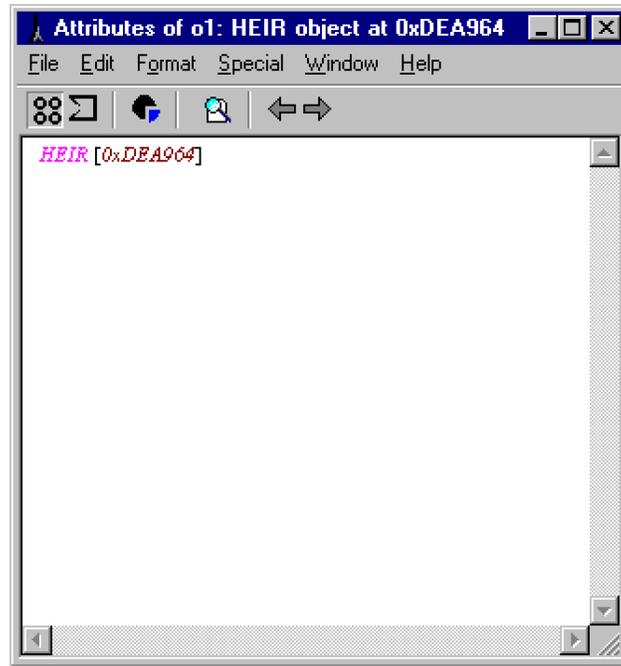
Exploring the Object Tool

The last tool to explore in this tutorial is the **Object Tool**, which displays in the upper-right window of the **Project Tool**. The **Object Tool** is used during execution and debugging and displays the contents of the current execution object — the one in which execution stopped. It contains one field for each attribute in the generating class.



As you can see, the active object is an instance of **TESTROOT** and contains three fields that correspond to the attributes **inv**, **o1** and **o2**. All three fields are of type reference; in other examples, you may encounter fields of basic types — boolean, integer, real and so on.

Although two of the references are void (**inv** and **o2**), **o1** (of type **HEIR**) contains the execution object **[0xDEA964]**. To view this object in a new **Object Tool**, hold down CTRL, and then right-click **[0xDEA964]** — again, the value differs on each computer.



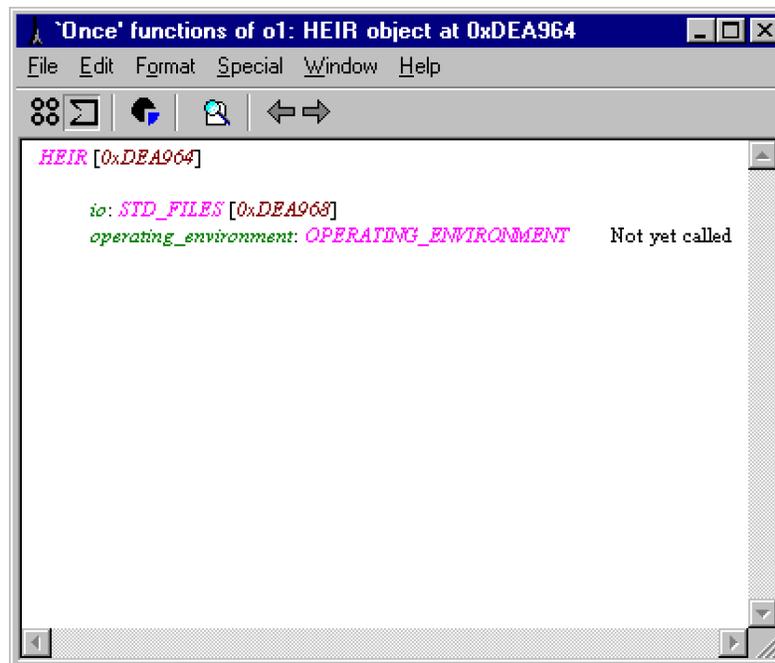
Since class **HEIR** has no attributes, the object contains no fields. However, since **HEIR** contains **once functions** — functions that only execute once; subsequent calls return the original value; useful for object sharing, since you can view the corresponding values.

Viewing values for once functions

To view values for once functions:

- On the **Object Tool** toolbar, click **Once functions** .

The **Object Tool** window contains the following information:



- **io** — the first once function; provides access to standard input and output; attached to an object of type **STD_FILES**; inherits from **GENERAL**: the class from which every class automatically inherits.
- **operating_environment** — not yet called, so no associated values; also inherits from **GENERAL**.

Other debugging facilities

There are other debugging facilities in the ISE Eiffel environment. Of particular interest is the ability to stop on an exception *before* the exception occurs. This notifies you of an impending exception (assertion violation, call applied to void reference, operating system signal or other) and allows you to use EiffelBench to analyze the cause of the exception, and the surrounding object structure.

One important property of the symbolic debugging mechanisms of EiffelBench is that they are not exclusive of other mechanisms. In other words, there is not a separate debugger or browser, nor are there separate debugging or browsing modes. At every point in your project, you have access to all debugging and browsing facilities in EiffelBench — you do not have to switch between different tools or modes.

1.19 Towards further study

You have now created and compiled your first EiffelBench project. This means that you have learned many of the basic EiffelBench procedures, enabling you to produce useful software.

The following chapters assist you towards that goal, as they provide more detailed information on the functionality of EiffelBench.

